MINI-CURSO DE **Febrero 2010**

JAVA

Profesores:
Albert Llastarri Rezola
Roberto Ramírez Vique

Autor:
Iván Párraga García

1 PRÓLOGO

Hola a todos, ante todo quisiera agradecer a Albert Llastarri su inestimable ayuda a la hora de realizar este manual de Java (especialmente en la parte de Swing) así como a los antiguos profesores del curso de los cuales he heredado algunos ejercicios y parte de la estructura del mismo y finalmente a todos los alumnos que han ido pasando por el curso y que gracias a sus comentarios han contribuido a mejorar esta documentación.

En esta tercera edición se han añadido nuevos componentes gráficos en el desarrollo de GUI's y se han revisado, corregido y aumentado todos los capítulos. A pesar de que he corregido todos los errores tipográficos y de contenido que se han ido localizando a lo largo de ya 3 cursos con este manual, el hecho de que soy humano y de que se van añadiendo cada cuatrimestre nuevos contenidos lo hace susceptible de contener errores. Se agradecerá que se notifiquen los mismos (abajo encontráis mi e-mail) para erradicarlos de próximas ediciones..

Todos los ejercicios del curso han sido desarrollados y comprobados por mí y al finalizar el curso se os entregará un CD-ROM con todo el material; además a lo largo del curso se irán colgando versiones correctas de los programas que vayamos haciendo.

He tratado de ilustrar todo el manual con ejemplos, gráficos y tablas. Al final del mismo he incorporado índices para acceder rápidamente a estos materiales así como un apéndice con direcciones web interesantes.

Al desarrollar el manual he tenido en cuenta las siguientes convenciones:

```
el texto con este formato es código
```

```
Este texto es intérprete de comandos
```

> **Ejercicio**
>
> **este texto es un ejercicio**

Bueno, pues por el momento esto es todo. Espero sinceramente que el curso os sea provechoso y siempre que tengáis cualquier tipo de duda no dudéis en preguntar.

Para cualquier comentario no dudéis en mandarme un e-mail a ivanp@jedi.upc.es.

El Autor
Iván Párraga García

2 ÍNDICE

3 INTRODUCCIÓN

Java es un leguaje de programación orientado a objetos desarrollado por SUN cuya sintaxis está basada en C++, por lo que todos aquellos que estén acostumbrados a trabajar en este lenguaje encontrarán que la migración a Java se produce de forma sencilla y podrán verlo como su evolución natural: un lenguaje con toda la potencia de C++ que elimina todas las estructuras que inducían a confusión y que aumentaban la complejidad del código y que cumple todos los requerimientos del tan de moda actualmente paradigma de programación orientada a objetos.

Java se vio catapultado a la fama de la mano del éxito que tuvo la WWW. Las páginas webs pueden considerarse como libros o revistas on-line, es decir, son contenidos que están publicados en la red por lo que el internauta se mueve y busca información; la web es pasiva, estática, no interacciona con el usuario. Por otro lado una aplicación o programa es fundamentalmente activo: interactúa con el usuario y responde en consecuencia. Java fue la solución para dotar a la web de mayor dinamismo: a través de los applets se pudieron insertar pequeños programas dentro de la página web.

Si bien en un principio la web fue el principal motor que dio a conocer a Java, en la actualidad Java es utilizado para programación de todo tipo de aplicaciones.

3.1 Breve historia de Java

A pesar de lo que pueda se pueda en un principio pensar, Java no surgió inicialmente como un lenguaje de programación orientado a la web. Los orígenes se remontan al año 1991 cuando Mosaic (uno de los primeros browsers[1]) o la World Wide Web no eran más que meras ideas interesantes. Los ingenieros de Sun Microsystems estaban desarrollando un lenguaje capaz de ejecutarse sobre productos electrónicos de consumo tales como electrodomésticos.

Simultáneamente James Gosling, el que podría considerarse el padre de Java, estaba trabajando en el desarrollo de una plataforma software barata e independiente del hardware mediante C++. Por una serie de razones técnicas se decidió crear un nuevo lenguaje, al que se llamó Oak, que debía superar algunas de las deficiencias de C++ tales como problemas relacionados con la herencia múltiple, la conversión automática de tipos, el uso de punteros y la gestión de memoria

El lenguaje Oak se utilizó en ciertos prototipos de electrónica de consumo pero en un principio no tuvo el éxito esperado dado que la tecnología quizás era demasiada adelantada a su tiempo. No obstante lo positivo de estos primeros intentos fue que se desarrollaron algunos de los elementos precursores de los actuales componentes Java; componentes tales como el sistema de tiempo de ejecución y la API[2].

[1] browser: navegador de páginas web como lo pueden ser Netscape Navigator, Opera o Internet Explorer
[2] API: Application Program Interface (Interfaz para la Programación de Aplicaciones)

En 1994 eclosionó el fenómeno web y Oak fue rebautizado como Java. En un momento de inspiración, sus creadores decidieron utilizar el lenguaje para desarrollar un browser al que se llamó WebRunner, que fue ensayado con éxito, arrancando en ese momento el proyecto Java/HotJava. HotJava fue un browser totalmente programado en Java y capaz así mismo de ejecutar código Java.

A lo largo de 1995 tanto Java, su documentación y su código fuente como HotJava pudieron obtenerse para múltiples plataformas al tiempo que se introducía soporte para Java en la versión 2.0 del navegador Netscape.

La versión beta 1 de Java despertó un inusitado interés y se empezó a trabajar para que Java fuera portable a todos los sistemas operativos existentes.

En diciembre de 1995 cuando se dio a conocer la versión beta 2 de Java y Microsoft e IBM dieron a conocer su intención de solicitar licencia para aplicar la tecnología Java, su éxito fue ya inevitable.

El 23 de enero 1996 se publicó oficialmente la versión Java 1.0 que ya se podía obtener descargándola de la web.

A principios de 1997 aparece la versión 1.1 mejorando mucho la primera versión.

Java 1.2 (Java 2) apareció a finales de 1998 incorporando nuevos elementos. Según Sun esta era la primera versión realmente profesional.

En mayo del 2000 se lanza la versión 1.3 del J2SE (Java 2 Standar Edition) y hace unos meses, en febrero de este año, se lanzó la versión 1.4 (la versión 1.4.1 es ya una beta).

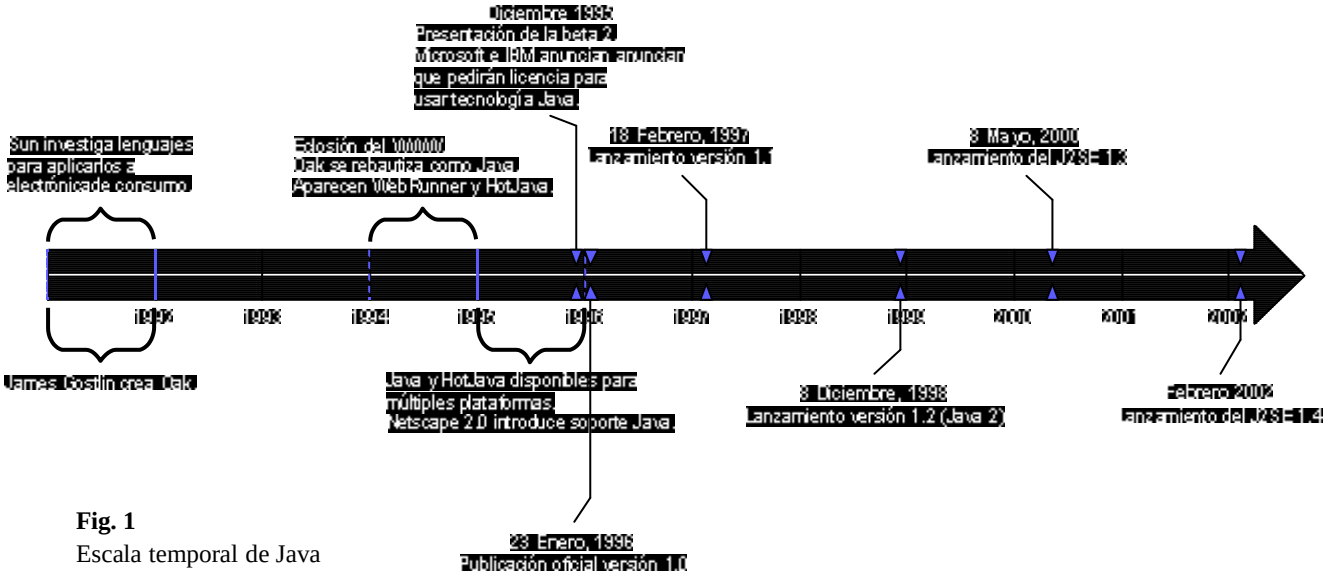
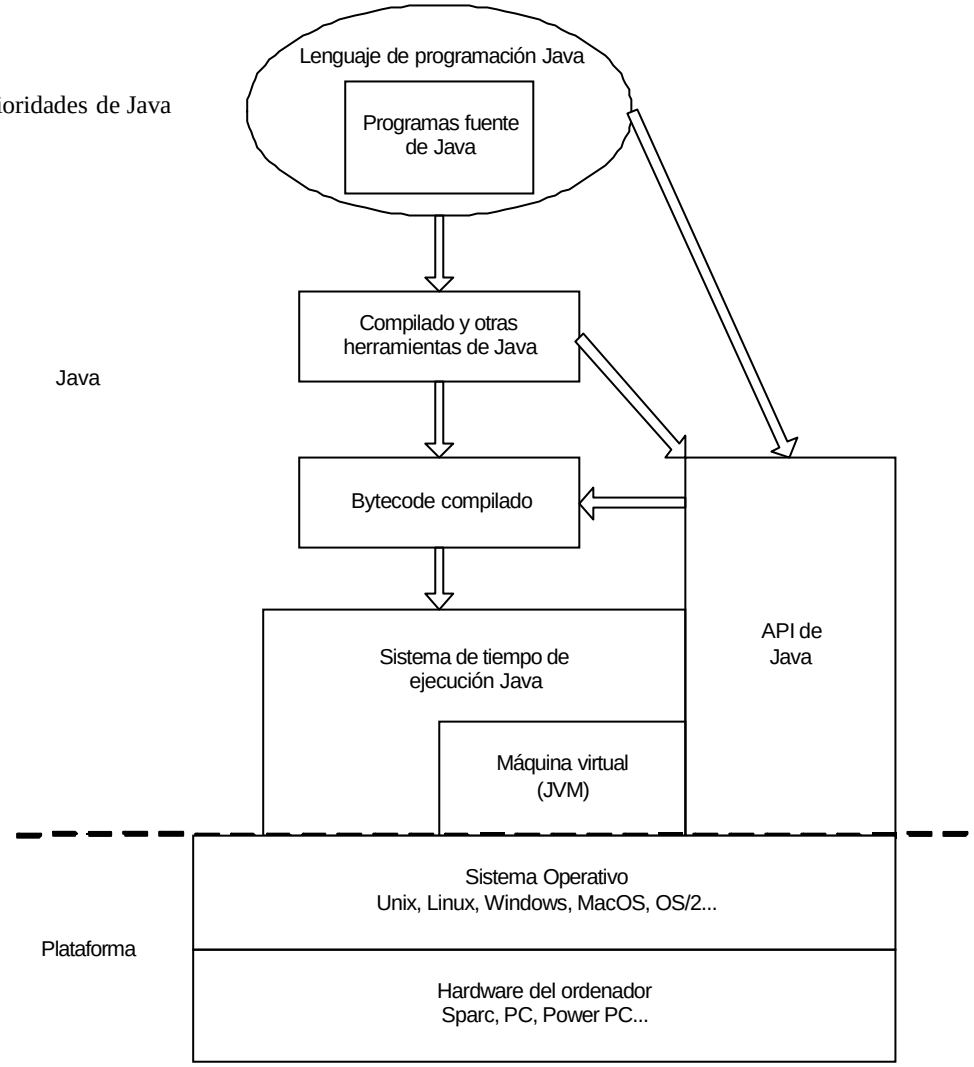


Fig. 1
Escala temporal de Java

3.2 ¿Qué es Java?

Java no es sólo un lenguaje de programación, Java es además un sistema de tiempo de ejecución, un juego de herramientas de desarrollo y una interfaz de programación de aplicaciones (API). Todos estos elementos así como las relaciones establecidas entre ellos se esquematizan en la figura 2.

Fig. 2
Las interioridades de Java



El desarrollador de software escribe programas en el lenguaje Java que emplean paquetes de software predefinidos en la API. Luego compila sus programas mediante el compilador Java y el resultado de todo ello es lo que se denomina bytecode compilado. Este bytecode es un fichero independiente de la plataforma que puede ser ejecutado por máquina virtual Java. La máquina virtual puede considerarse como un microprocesador que se apoya encima de la arquitectura concreta en la que se ejecuta, interactuando con el sistema operativo y el hardware, la máquina virtual es por tanto dependiente de la plataforma del host pero no así el bytecode. Necesitaremos tantas máquinas virtuales como plataformas posibles pero el mismo bytecode podrá ejecutarse sin modificación alguna sobre todas ellas.

Así pues, las claves de la portabilidad de Java son su sistema de tiempo de ejecución y su API. Los potentes recursos para el trabajo de ventanas y redes incluidos en la API de

Java facilitan a los programadores el desarrollo de un software que resulte a la vez atractivo e independiente de la plataforma.

Como se ha dicho Java es un lenguaje de programación orientado a objetos, pero veamos en más detalle qué es lo que nos ofrece. Un buena descripción del mismo puede encontrarse en uno de los primeros documentos que aparecieron sobre el lenguaje, el "White Paper" que a pesar de haber sido escrito por sus desarrolladores es bastante fiel a la realidad:

Simple. Java, como todo lenguaje de programación de propósito general, no es sencillo de usar y menos de dominar. Por tanto debemos definir su simplicidad en función de algo. Si nos basamos en la dificultad de programar en C++ (lenguaje tomado como origen por los diseñadores de Java), entonces sí podemos afirmar que el lenguaje es simple.

Las principales simplificaciones consisten en no necesitar tener ficheros de cabeceras, la ausencia de punteros, no tener que gestionar la memoria y disponer de un extenso surtido de librerías que nos facilita la faena cuando tenemos que enfrentarnos a programas de cierta complejidad. También tenemos muchas menos estructuras de programación y habitualmente más sencillas.

Orientado a objetos. Java es, efectivamente, un lenguaje plenamente orientado a objetos. Esto es una ventaja especialmente en las etapas de especificación y diseño, así como a la hora de tratar con todos los aspectos de la programación en entornos gráficos. Respecto a C++, las mayores diferencias son la forma diferente de ligar invocaciones polimórficas a métodos y la manera distinta de resolver la herencia múltiple.

Distribuido. Una de las líneas principales que marcaron el diseño de Java fue su orientación a usos relacionados con Internet. Cualquiera que haya intentado programar alguna aplicación Internet en otro lenguaje se dará cuenta de la simplicidad de Java en estos menesteres. Incluso la programación de CGI's resulta mucho más simple. También es bastante común la ejecución de parte (o toda) la aplicación por parte del cliente, con los llamados *applets*.

Robusto. Un programa robusto es aquél que es resistente a errores, es decir, aquél que ante un determinado error, no detiene su ejecución bruscamente. Java simplifica la labor de programar software robusto. Una de las ventajas en este sentido es el hecho de ser un lenguaje fuertemente tipificado, aunque la que más nos facilitará el trabajo es la de no tener que preocuparnos de la gestión de la memoria gracias a que no tiene punteros (aunque podemos beneficiarnos de sus ventajas, como listas encadenadas, etc.) y dispone de un *garbage collector* que nos evita el trabajo de ir liberando la memoria (con los errores que suele comportar).

La robustez de los programas Java también se ve beneficiada gracias a que muchos errores que normalmente encontramos en tiempo de ejecución Java los detecta en tiempo de compilación. También podemos recoger los errores que se produzcan en tiempo de ejecución gracias al mecanismo de excepciones, donde podremos dar soluciones alternativas a los problemas que puedan suceder en la ejecución de nuestro programa.

Seguro. Puesto que se pretende usar Java en entornos en red y distribuidos se han cuidado mucho los aspectos relacionados con seguridad. Parece difícil que una

aplicación Internet hecha con Java pueda dañar la integridad del sistema cliente accediendo a zonas de memoria o del sistema de ficheros no permitidas, pero este es un punto en el que siempre pueden surgir agujeros.

Arquitectura neutral. El compilador genera un fichero objeto con un formato independiente de la arquitectura. Esto permite que el código compilado sea ejecutable en todos los procesadores para los que exista un *run time system* Java (llamado Máquina Virtual Java). El compilador logra esto generando instrucciones *bytecode*, que no son de ninguna arquitectura en particular, sino que están pensadas para ser fácilmente traducidas *on the fly* a cualquier lenguaje máquina existente, y por tanto interpretadas.

Esto supone que un programa hecho en Java y compilado a *bytecode* puede funcionar en diferentes plataformas hardware/software y comportarse exactamente de la misma forma (en programación gráfica con AWT con ligeras diferencias).

Portable. Además de generarse un código independiente de la arquitectura se han cuidado al máximo todos los aspectos relacionados con portabilidad. Los tipos de datos ocupan estrictamente lo mismo en cualquier plataforma (a diferencia de C/C++ donde un int puede tener 16 o 32 bits según la implementación), y se codifica todo en *big endian*.

Intepretado. Java no genera un ejecutable diferente cada vez que compilamos en una plataforma distinta sino que, independientemente de la plataforma, se generará el mismo código intermedio (*bytecode*), que después será ejecutado en cualquier plataforma por un intérprete específico para la plataforma.

Por desgracia, el hecho de que Java sea un lenguaje interpretado implica necesariamente que sea más lenta su ejecución. Además también hay que tener en cuenta que el proceso de compilación es lento porque busca la máxima optimización posible y tiene que comprobar más cosas que la mayoría de lenguajes. De todos modos en las últimas versiones de Java se han conseguido resultados espectaculares para tratarse de un lenguaje interpretado.

Alto rendimiento. Un lenguaje interpretado acostumbra a ser entre 20 y 100 veces más lento que el mismo programa compilado y ejecutado. Teniendo esto en cuenta, el rendimiento de Java es más que aceptable, pero en valor absoluto, hablar de "alto rendimiento" es una ligera imprecisión.

Multiflujo. La programación multiflujo (*multithreading*) en Java ofrece unos resultados muy buenos, especialmente si los comparamos con otros lenguajes. En parte es debido a que ofrece unas librerías específicas que nos simplifican mucho el trabajo. Se aprovechan también las ventajas de los sistemas multiprocesadores. Por contra cabe destacar que el comportamiento en cuanto a *scheduling* puede variar de una plataforma a otra, en detrimento de la independencia de plataforma.

Dinámico. Java está diseñado para adaptarse a entornos en evolución. En este sentido las librerías pueden añadir nuevos métodos y atributos a sus clases sin afectar a los clientes. Es posible cargar nuevas clases en tiempo de ejecución y consultar los métodos que estas tienen disponibles, etc

.

4 J2SDK, EL ENTORNO DE PROGRAMACIÓN

Como se ha mencionado ya, Java no es sólo la definición del lenguaje, sino que también es todo el conjunto de herramientas que constituyen el entorno de programación así como el entorno de ejecución. El J2SDK (Java 2 Software Develpment Kit), incluye ambos.

Sun cambió la nomenclatura con la que se refirió a sus versiones de Java al pasar a la versión 1.2, así pues cuando se habla de Java 2 nos estamos refiriendo a la versión 1.2 o superior de Java. El entorno de programación y de ejecución también fue rebautizado. En un principio recibía el nombre de JDK (Java Development Kit) y actualmente recibe el ya mencionado nombre de J2SDK. La última versión estable a fecha de hoy es la versión J2SDK 1.4 (con la que trabajaremos en este curso) y está a punto de lanzarse la 1.4.1.

Quizás todo este conjunto de nomenclaturas induzca a confusión. A continuación presente un cuadro que intenta aclarar todo esto para la versión 1.2 de Java (siguiendo versiones posteriores el mismo esquema):

Tabla 1. Explicación de las versiones Java

Nuevo Nombre	Antiguo Nombre	Explicación
Java™ 2 Platform, Standard Edition, v 1.2 (J2SE)	Ninguno	Es la plataforma abstracta de Java 2 que constituye la tecnología y/o entorno de descrito en las especificaciones de Sun versión 1.2
Java™ 2 SDK, Standard Edition, v 1.2 (J2SDK)	JDK™ version 1.2	Es la implementación de Sun de la J2SE versión 1.2, incluyendo tanto el kit de desarrollo de software como el entorno de ejecución.
Java™ 2 Runtime Environment, Standard Edition, v 1.2 (J2RE)	JRE 1.2	Es la implementación de Sun del entorno de ejecución correspondiente a la versión 1.2

Finalmente mencionar que la documentación de la API no está incluida en el J2SDK y es necesaria descargarla a parte. Esta documentación es la herramienta básica de consulta para todo desarrollador de Java por lo que es altamente recomendable (por no decir totalmente necesario) que se descargue.

Todo este software y documentación puede obtenerse directamente de la web de Sun (http://java.sun.com/j2se/).

4.1 Instalación

La última versión oficial de Sun para plataforma PC/Windows tiene, literalmente, los siguientes requermientos: "un Pentium 166Mhz o mejor, con al menos 32 Mb de RAM física para ejecutar aplicaciones con interfaz gáfico. 48 Mb de RAM recomendados para applets ejecutándose desde un navegador usando el plug-in de Java. Ejecutarlos con menos memoria puede causar excesivo swap a disco lo cual tendría un serio efecto en su correcta visualización. Muchos programas pueden necesitas más RAM para una ejecución adecuada. Debería tener 70 Mb libres de disco duro antes de intentar instalar el J2SDK. Si también desea instalar la documentación, necesitará unos 120 Mb adicionales."

El J2SDK generalmente se obtiene en un archivo comprimido dependiendo el fomato del mismo de la plataforma a la que esté destinado, así pues puede tratarse de un archivo .zip un .tar.gz o incluso un .exe autoinstalable. En cualquier caso lo primero que hay que hacer es descomprimirlo respetando la estructura de directorios que ya viene establecida.

A continuación sólo falta configurar dos variables de entorno para poder comenzar a trabajar. Tendremos que añadir la variable CLASSPATH para que apunte al fichero src.jar contenido en el directorio raíz de la instalación y también para que apunte al directorio actual. Esta variable indica al conjunto de herramientas del J2SDK donde tiene que ir a buscar el código fuente de las clases que constituyen nuestra aplicación. Finalmente habrá que modificar la variable PATH para que apunte a los binarios que constituyen dichas herramientas, es decir, para que apunte al subdirecotrio bin que cuelga del raíz de la instalación.

La configuración de estas variables de entorno se hace de manera distinta dependiendo de la plataforma en la que estemos trabajando. Bajo Windows 9x/Me suelen añadirse estas líneas al autoexec.bat (suponiendo que el entorno se haya instalado en C:\Java):

```
SET CLASSPATH=C:\java\src.jar;.
SET PATH=%PATH%;C:\java\bin
```

En Windows NT/2000/XP estas variables de entorno pueden modificarse o bien por la línea de comandos o bien a través de Mi Pc -> Propiedades -> Avanzado -> Variables de entorno.

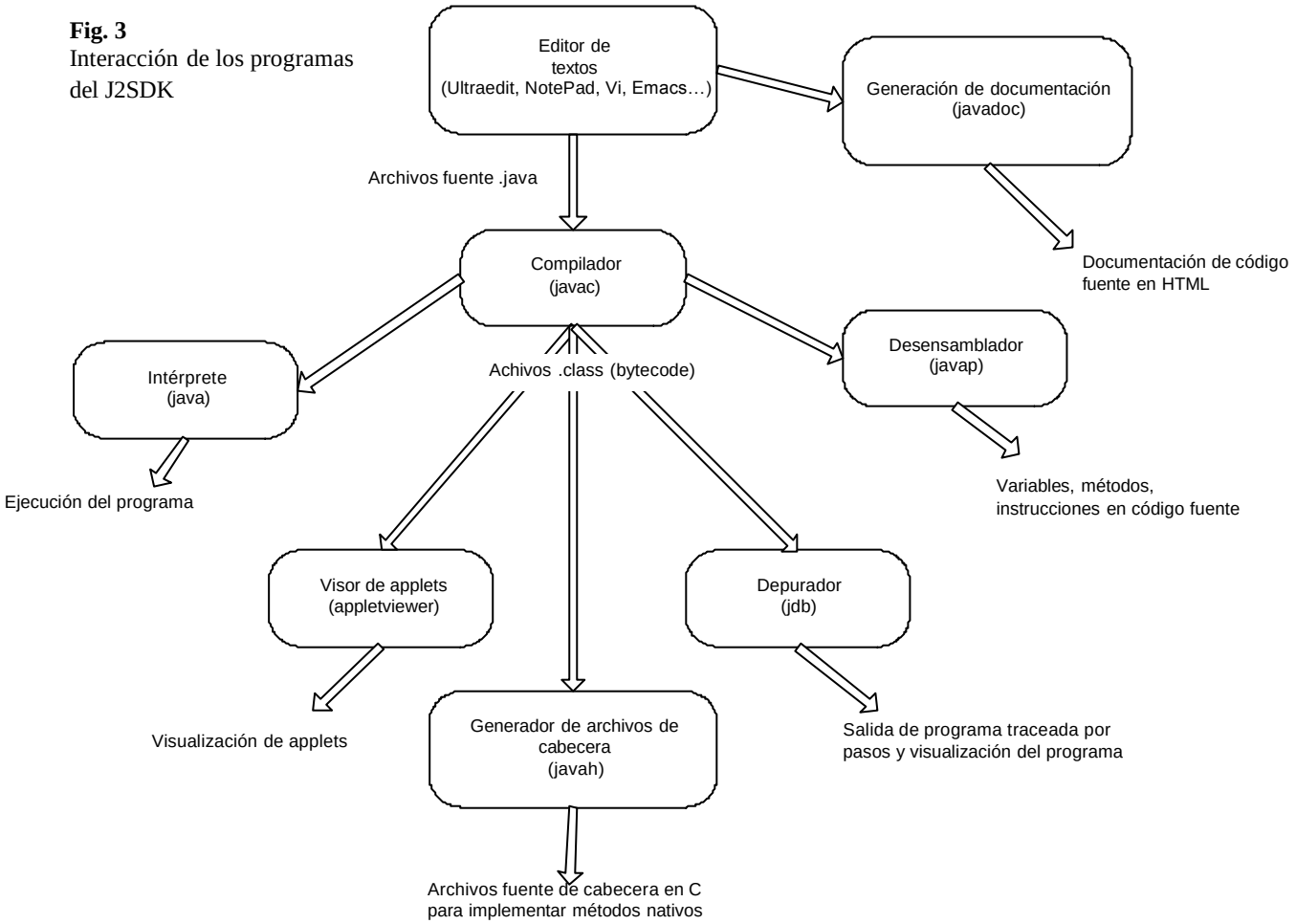
En sistemas Linux dependerá del tipo de shell[3] que estemos ejecutando, pero igualmente se pueden incorporar la líneas que sean necesarias a los scripts de inicio para no tener que modificar las variables en cada sesión.

[3] shell: intérprete de comandos

4.2 ¿Qué aplicaciones hay en el J2SDK?

El entorno de J2SDK es de tipo línea de comando. Constituye todas las herramientas necesarias para desarrollar aplicaciones Java, así pues consta del compilador/linkador, del intérprete de aplicaciones, de un debugger, de un visor de applets y de un programa de generación automática de documentación, entre otros. Existen entornos de desarrollo Java ajenos a Sun como pueden ser Kawa, SuperCD, Jbuilder, Visual Café... El J++ de Microsoft no es un producto Java ya que no era compatible con las especificaciones de Sun.

Fig. 3
Interacción de los programas del J2SDK



4.2.1 Compilación: javac

Para generar el archivo .class, que es el bytecode que recibe el entorno de ejecución, se usa el compilador javac que recibe el código fuente en un fichero con extensión .java.

Para compilar una aplicación basta con compilar la clase principal, aquella donde está el método main, y después de forma automática se va llamando para compilar todas las clases que necesite. Estas clases deben ser accesibles a través de la variable de entorno CLASSPATH.

Vamos a hacer un pequeño ejemplo para probar el compilador. No os preocupéis si no entendéis la sintaxis del programa, todo se irá aclarando a lo largo del curso. Abrid vuestro editor preferido y copiad las siguientes líneas y a continuación guardad el fichero como *HolaMundo.java*

```
public class HolaMundo
{
      public static void main(String[] args)
      {
            System.out.println("Hola mundo!");
      }
}
```

Una vez hayáis hecho esto ejecutad el compilador de la siguiente manera:

```
> javac HolaMundo.java
```

si todo ha ido bien en el mismo directorio debería haber aparecido el correspondiente *HolaMundo.class.* Si hubiéramos cometido un error de sintaxis al editar el fichero, el compilador nos habría informado de tal situación indicándonos en qué clase se ha producido el error, en qué número de línea del fichero correspondiente y de qué tipo de error se trata. Por ejemplo si hubiéramos copiado por error lo siguiente:

```
public class HolaMundo
{
      public static void main(String[] args)
      {
            Systum.out.println("Hola mundo!");
      }
}
```

habríamos obtenido el siguiente mensaje de error:

```
HolaMundo.java:5: cannot resolve symbol
symbol  : class out
location: package Systum
                Systum.out.println("Hola mundo!");
                      ^
1 error
```

Puede variarse el comportamiento del compilador mediante los diversos parámetros que se le pueden pasar. Para ver una lista de los mismos basta con invocarlo sin indicarle ningún fichero fuente.

4.2.2 Ejecución de aplicaciones: java

Ahora que ya hemos compilado nuestro flamante *HolaMundo.java* y estamos ansiosos por ver el resultado de nuestro primer programa, usaremos el intérprete Java para ejecutarlo. Nada más sencillo que teclear el comando:

```
> java HolaMundo
```

Notad que para ejecutar nuestro programa debemos tener el .class correspondiente y que tenemos que pasar el nombre de la clase principal sin ningún tipo de extensión. Tened en cuenta también que Java es sensible al tamaño, lo cual quiere decir que debéis respetar mayúsculas y minúsculas a la hora de ejecutar aplicaciones. Si hubiéramos ejecutado el comando así:

```
> java holamundo
```

el entorno nos hubiera indicado que no ha sido capaz de encontrar la clase correspondiente con un mensaje similar a:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
holamundo (wrong name
: HolaMundo)
        at java.lang.ClassLoader.defineClass0(Native  Method)
        at java.lang.ClassLoader.defineClass(Unknown  Source)
        at java.security.SecureClassLoader.defineClass(Unknown
Source)
        at java.net.URLClassLoader.defineClass(Unknown  Source)
        at java.net.URLClassLoader.access$100(Unknown  Source)
        at java.net.URLClassLoader$1.run(Unknown  Source)
        at java.security.AccessController.doPrivileged(Native
Method)
        at java.net.URLClassLoader.findClass(Unknown  Source)
        at java.lang.ClassLoader.loadClass(Unknown  Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown
Source)
        at java.lang.ClassLoader.loadClass(Unknown  Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown
Source)
```

4.2.3 Ejecución de applets: appletviewer

Los applets son miniaplicaciones Java con interfaz gráfica que se insertan en páginas web. Para verlos es necesario un navegador que soporte Java o usar esta mini aplicación.

Para probarlo vamos a ejecutar uno de los múltiples applets de demostración que se instalan con el J2SDK. Suponiendo que este esté instalado en C:\Java teclearemos el siguiente comando para visualizar el applet:

```
> appletviewer c:\java\demo\applets\TicTacToe\example1.html
```

con lo que se ejecuta nuestro applet, una versión del conocido tres en raya.

Fig. 4
Applet TicTacToe

Hay que tener en cuenta que un applet sólo puede visualizarse si está insertado en una página web por eso para poder ejecutarlo le hemos indicado al appletviewer que "ejecute" la página web y no la clase correspondiente al mismo.

Al igual que con el compilador y el intérprete, para poder ver los parámetros admitidos por el visor de applets sólo hay que ejecutar el comando sin indicarle ningún URL.

4.2.4 Corrección de errores: el **debugger** *jdb*

El J2SDK incluye un *debugger* para poder depurar el programa y localizar de forma mas rápida y precisa los errores. Por desgracia `jdb` no es una de las experiencias más satisfactorias de esta vida. Para ejecutar el *debugger*, además, es necesario tener configuradas las opciones de red en la máquina.

El procedimiento para usar el *debugger* sobre una clase consiste en compilar primero con la opción `-g` y luego usar el *debugger* en lugar del intérprete:

```
> javac -g miClase. java
> jdb miClase
```

`jdb` es un *debugger* del tipo "línea de comando" bastante rudimentario, lo que hace que sea bastante desaconsejable usarlo. Para localizar los errores en un programa acaba siendo más recomendable usar el antiguo y efectivo método de insertar chivatos en el fragmento de código susceptible de tener algún error o simplemente preguntarle al profesor.

5 ESTRUCTURAS BÁSICAS DE PROGRAMACIÓN

En Java todos los programas se construyen a partir de clases, dentro de esas clases encontramos declaraciones de variables (instrucciones atómicas) y procedimientos o funciones (conjuntos de instrucciones).

En este capítulo vamos a ver cómo funcionan las instrucciones fundamentales de Java. Para los programadores habituales de C/C++ resultará muy sencillo adaptarse a las instrucciones de Java ya que en la mayoría de los casos son idénticas.

5.1 Comentarios

Los comentarios son cadenas de texto que el programa usa para entender y hacer inteligible su código a otros. Los comentarios son ignorados por el compilador. Java tiene tres tipos de comentarios como los que ilustraremos en el siguiente programa:

```
01 /* ClasicoHolaMundo.java */
02
03 public class ClasicoHolaMundo
04 {
05          /**
06          * Clásico Programa "Hola Mundo!"
07          */
08
09          public static void main(String[] args)
10          {
11                // escribe por la salida estándar
12                System.out.println("Hola mundo!");
13          }
14 }
```

Tenemos el comentario tipo C que empieza con /* y termina con */ como en la línea 1, los comentarios tipo C++ que comienzan con // y llegan hasta el final de una línea como en la línea 11 y los comentarios de apoyo documental que comienzan con /** y que terminan con */ como en las líneas de la 5 a la 7. Estos últimos comentarios se usan para, junto con la aplicación javadoc , generar automáticamente la documentación de la clase en html. Los comentarios no pueden anidarse y no pueden aparecer dentro de cadenas y literales de carácter. Recapitulando:

```
/* Esto es un comentario estilo C que puede ocupar varias líneas */

// Esto es un comentario que se extiende hasta el final de la línea

/** Esto es un comentario de javadoc que puede extenderse entre varias
líneas */
```

5.2 Identificadores

Los identificadores se usan para nombrar y referirnos a las entidades del lenguaje Java, entidades tales como clases, variables o métodos. Como ya se ha dicho anteriormente, Java es sensible a la diferenciación de mayúsculas y minúsculas, así pues una variable *contador* no es la misma que otra *Contador* y pueden, aunque no sea aconsejable, coexistir sin problemas. Un identificador tampoco puede ser igual a una palabra reservada.

5.3 Palabras reservadas

Las palabras reservadas por el lenguaje Java son las que se muestran a continuación. Al final de este curso conoceremos el significado de la mayoría de ellas:

abstract	default	if	outer	this
assert	do	implements	package	throw
boolean	double	import	private	throws
break	else	inner	protected	transient
byte	extends	instanceof	public	try
case	final	int	rest	var
cast	finally	interface	return	void
catch	float	long	short	volatile
char	for	native	static	while
class	future	new	super	
const	generic	null	switch	
continue	goto	operator	synchronized	

5.4 Tipos de datos primitivos

Java es un lenguaje fuertemente tipificado, lo que quiere decir toda variable debe ser declarada de un tipo. De los ocho tipos primitivos, hay seis numéricos (cuatro enteros y dos en coma flotante), otro es el carácter y el último es el booleano. La portabilidad de Java garantiza todos los tipos tendrán el mismo tamaño independientemente de la plataforma.

5.4.1 Enteros

Tabla 1. Tipos enteros

Tipo	Tamaño (en bytes)
int	4
short	2
long	8
byte	1

5.4.2 Tipos en coma flotante

Tabla 2. Tipos en coma flotante

Tipo	Tamaño (en bytes)	Cifras significativas
float	4	7
double	8	15

Los tipos *float* y *double* representan números de coma flotante IEEE754 de 32 y 64 bits. Los números float tiene el sufijo f o F. Los números dobles tienen d o D. Si no se proporciona ningún sufijo se asume el tipo por defecto double. Los números de coma flotante pueden escribirse en cualquiera de los cuatro formatos siguientes (siendo el sufijo opcional tal como se acaba de describir):

Tabla 3. Formato tipos en coma flotante

Formato	Ejemplos
dígitos . dígitosOpcionales exponente opcional sufijo	43.32 43.32e3f
. dígitos exponenteOpcional sufijo	.34E-4F .22d
dígitos exponente sufijo	3e0D 45E-7
NaN	

El valor especial NaN (Not a Number) se utiliza para representar resultados de operaciones matemáticas que obtienen un valor indefinido tales como, por ejemplo, una división por cero.

5.4.3 Caracteres: char

Los caracteres se almacenan en el tipo *char*. Java utiliza el código Unicode de 16 bits (diferenciándose de la mayor parte de lenguajes clásicos, como C/C++, que utilizan ASCII de 8 bits). Unicode es un superconjunto de ASCII, es decir ASCII está contenido en Unicode, pero este último proporciona muchísimos caracteres más (los dos bytes permiten tener 2^{16}=65.536 caracteres diferentes frente a los 2^{8}=256 caracteres del ASCII extendido).

Los caracteres se encierran entre comillas sencillas (‘’) y no dobles (“”). Los caracteres de escape, al igual que en C/C++, se preceden de la barra invertida (\). Tenemos lo siguiente códigos de escape:

Tabla 4. Códigos de escape

Código de escape	Carácter	Cifra hexadecimal
\b	retroceso	\u0008
\t	tabulación	\u0009
\n	avance de línea	\u000a
\f	avance de papel	\u0006
\r	retroceso de carro	\u000d
\”	comillas normales	\u0022
\’	comillas sencillas	\u0027
\\	barra invertida	\u005c

Podemos utilizar los caracteres como ‘a’, ‘$’ o ‘b’, pero también podemos refenciarlo directamente por su código Unicode en hexadecimal de la siguiente manera \u0061, \u0024 y \u0062 respectivamente. Notad que el código Unicode que representa un carácter existente en el código ASCII es el mismo precedido de un byte en 0, es decir ‘a’ es el número 61h del código ASCII y es el número 0061h del código Unicode.

5.4.4 Booleanos: boolean

A diferencia de C/C++ los valores cierto y falso que se utilizan en expresiones lógicas, no se representan con un entero que toma los valores 1 y 0, sino que existe un tipo destinado a tal efecto, el tipo *boolean* que toma valores *true* y *false*.

5.5 Declaraciones de variables

Una variable es una estructura que se referencia mediante un identificador que nos sirve para almacenar los valores que usamos en nuestra aplicación. Para usar una variable debemos declararla previamente de un tipo. Veamos algunos ejemplos:

```
boolean b;
int numero
float decimal=43.32e3f
int contador=0;
char c='a';
```

como vemos la forma de declarar variables es de la forma:

```
tipoVariable identificadorVariable [= valorInicial];
```

donde primero indicamos el tipo al que pertenece la variable, a continuación el identificador o nombre de la variable, opcionalmente un valor inicial y finalmente acabamos con un ;. A diferencia de otros lenguajes, Java permite declarar las variables en cualquier parte de un bloque de instrucciones (no tiene porqué ser al principio).

5.6 Constantes

El concepto de constante no puede entenderse estrictamente en el paradigma de orientación a objetos y por ello en Java no existen las constantes propiamente dichas. Esto se explicará con más detalle cuando pasemos al capítulo de orientación a objetos, por el momento haced un acto de fe y creeros que lo más parecido que tenemos a una constante es una *constante de clase no modificable* que se declara de la siguiente manera:

```
static final tipo indentificador = valor;
```

por ejemplo:

```
static final float pi = 3.141592F;
```

5.7 Asignaciones

El operador de asignación, como ya hemos visto en las inicializaciones de variables, es el "=", por ejemplo:

```
bool javaEsGenial;
javaEsGenial=true;

int edad;
edad=22;
```

Es posible hacer asignaciones entre tipos numéricos diferentes. Esto es posible de una manera implícita en el sentido de las flechas (no puede haber perdida de información):

```
byte -> short -> int -> long -> float -> double
```

o explícita cuando tiene ser al revés mediante un *casting*, siendo responsabilidad del programador la posible pérdida de información ya que Java no producirá ningún error. Por ejemplo:

```
float radio;
radio = 5;  // no es necesaria la conversion explícita de int a float

int perimetro;
perimetro = radio * 2 * PI;   // error! no se puede guardar en un int
                              // el resultado de una operación con float

perimetro = (int) (radio * 2 * PI)  // ahora es correcto porque el
                                    // casting fuerza la conversion
```

El *casting* de carácter a un tipo numérico también es posible, aunque no demasiado recomendable. En cambio no es posible la conversión entre booleanos y tipos numéricos.

5.8 Strings

Ante todo dejar claro que los strings no son tipos primitivos en Java. Son una clase implementada en la biblioteca estándar aunque con ciertas funcionalidades que hacen que su uso sea comparable al de los tipos primitivos en ciertos aspectos.

Podemos declarar e inicializar *strings* como:

```
String st = "";          // String vacío
String st1 = "Hola";
String st2 = "cómo estas?";
String st3 = st1 + " " + st2; // st3 vale "Hola cómo estas?"
```

La cadena `st3` contiene una concatenación de las cadenas st1, un espacio en blanco y `st2`. Para concatenar `String`s basta con operarlos mediante el símbolo "+". Cuando concatenamos un `String` con algo que no lo es, este valor es automáticamente convertido a `String`.

```
String st = "numero: " + 3;   // st vale "numero: 3"
```

Además de la concatenación, también podemos usar otras operaciones de la clase `String` para trabajar con ellos. Algunos ejemplos pueden ser (veremos la forma general de llamar operaciones que actúan sobre objetos en el próximo capítulo):

```
String st = "String de ejemplo";
String st2 = "String de ejemplo";
```

`st.length()`. Devuelve la longitud de la cadena `st`.

`st.substring(a, b)`. Devuelve la subcadena `a` partir de la posición a (incluida) hasta la `b` (no incluida).

`st.charAt(n)`. Devuelve el carácter en la posición `n`.

`st.equals(st2)`. Devuelve un booleano que evalúa a cierto si los dos `String` `st` y `st2` son iguales, en este caso valdrá `true`.

Cabe destacar que cuando consideramos posiciones dentro de un `String`, si la longitud es n, las posiciones válidas van de la 0 a la n-1. Otro aspecto a tener en cuenta es que una variable de tipo `String` es un puntero a una zona de memoria (ya veremos que ocurre en general con todos los objetos). Así pues, (si no hay ciertas optimizaciones por parte del compilador) la comparación `st == st2` evalúa a falso.

5.9 Arrays

Java ofrece *arrays* parecidos a los de C/C++, pero implementados como objetos de primer orden. Los componentes de un *array* pueden ser tipos primitivos o referencias a objetos. En la declaración de un *array* se omite el tamaño, que se especifica en la inicialización. A diferencia de C/C++ no obtenemos espacio en memoria con sólo declarar una variable de tipo *array*: hay que inicializarla, y se puede hacer de diversas formas. Una vez inicializado el tamaño del *array* no puede ser modificado.

```
int[] nums;
nums = new int[100];
```

o lo que es lo mismo:

```
int[] nums = new int[100];
```

son declaraciones e inicializaciones de *arrays* de 100 enteros. El operador `new` es el mismo que se usa para pedir memoria para cualquier objeto, en este caso se pide espacio contiguo para 100 enteros.

Ahora podemos acceder a los elementos del *array* con `nums[i]`, donde `i` toma un valor entre 0 y 99. Es posible crear matrices multidimensionales:

```
int[][] matriz = new int[4][4];
```

También podemos inicializar los *arrays* con valores:

```
String[] semana = {"lunes", "martes", "miércoles", "jueves","viernes",
"sábado", "domingo"};
```

Se puede consultar la longitud de un *array* con el atributo `length`:

```
int diasSemana = semana.length;           // diasSemana vale 7
```

5.10 Operadores

Java define operadores aritméticos, relacionales, lógicos de manipulación de bits, de conversión de tipo, de clase, de selección y de asignación. No os preocupéis si ahora no queda claro el significado de alguno de ellos

Tabla 5. Operadores Java

Tipo de operador	Operador	Descripción	Ejemplo
Aritmético	+	Suma	a+b
	-	Resta	a-b
	*	Multiplicación	a*b
	/	División	a/b
	%	Módulo	a%b
Relacional	>	Mayor que	a>b
	<	Menor que	a<b
	>=	Mayor o igual que	a>=b
	<=	Menor o igual que	a<=b
	!=	Diferente	a!=b
	==	Igual	a==b
Lógico	!	No	!a
	&&	Y	a&&b
	||	O	a||o
Manipulación de bits	~	Complemento	~a
	&	Y bit a bit	a&b
	|	O bit a bit	a|b
	^	O exclusivo bit a bit	a^b
	<<	Desplazamiento a la izda.	a<<b
	>>	Desplazamiento a la dcha.	a>>b
	>>>	Desplazmiento a la dcha. rellenando ceros	a>>>b
Asignación	=	Asignación	a=b
	++	Incremento y asignación	a++
	--	Decremento y asignación	a--
	+=	Suma y asignación	a+=b
	-=	Resta y asignación	a-=b
	*=	Multiplicación y asignación	a*=b
	/=	División y asignación	a/=b
	%=	Módulo y asignación	a%=b
	|=	O y asignación	a|=b
	&=	Y y asignación	a&=b
	^=	O exclusiva y asignación	a^=b
	<<=	Desplazamiento a la izda. y asignación	a<<=b
	>>=	Desplazamiento a la dcha. y asignación	a>>=b
	>>>=	Desplazamiento a la dcha. rellenando ceros y asignación	a>>>=b
Conversión de tipo (casting)	(tipo)	Convertir tipo	(char)b
Instancia	instanceof	¿Es instancia de clase)	a instanceof b

Reserva	new	Crear un nuevo objeto de una clase	new A()
Selección	?:	Si ... entonces selección	a?b:c

La precedencia de los operadores, esto es, cuales se evaluarán antes a falta de paréntesis que indiquen lo contrario es la siguiente de arriba a abajo y de izquierda a dereha:

Tabla 6. Precedencia de los operadores

+ Precedencia - Precedencia	

`[ ] . ( )` (en llamada a función)
`¡ ~ ++ -- +`(unario) `-`(unario) `( )` (en *cast*) `new`
`/ %`
`+ -`
`<< >> >>>`
`< <= > >= instanceof`
`== ¡=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= *= /= %= &= |= ^= <<= >>= >>>=`
	+ Precedencia
	
	- Precedencia

5.11 Estructuras de control de flujo

Antes de meternos a fondo con las estructuras de control de flujo debemos tener claro qué es un *bloque*. Los bloques consisten en secuencias de declaraciones e instrucciones de variables locales. Se escribe como sigue:

```
{ bloqueCuerpo }
```

donde bloqueCuerpo es una secuencia de declaraciones e instrucciones de variables locales. Un bloque también puede consistir en una instrucción única sin tener que estar entre llaves.

La sentencia vacía consiste en un solo punto y coma (;) y no realiza proceso alguno.

5.11.1 Condicionales

Permiten desviar el flujo de ejecución por una rama u otra dependiendo de la evaluación de una condición. Existen dos estructuras condicionales, el *if*::

```
if (condición1)
    {bloque1}
[else if (condición2)
    {bloque2}
...]
[else
    {bloqueN}]
```

que puede tener una o más ramas que evalúen condiciones y opcionalmente un *else* cuyo bloque se ejecutará sólo si no se ha cumplido ninguna de las condiciones anteriores. Toda condición (en estas estructura y en adelante) debe evaluarse a un valor booleano no

pudiéndose utilizar valores enteros como en C/C++. La otra estructura condicional es el *switch* que permite elegir un bloque en función del valor de una variable de referencia:

```
switch (variableReferencia)
{
    case valor1:
           {bloque1}
    [case valor2:
           {bloque2}
    ...]
    [default:
           {bloqueN}
    ]
}
```

variable Referencia sólo puede ser una expresión que evalúe a los tipos primitivos enteros o char. Se evalúa la expresión y sucesivamente se va comprobando que coincida con alguno de los valores, en caso de que así sea se ejecuta el bloque correspondiente. Hay que tener en cuenta que la última instrucción de cada bloque debe ser un *break* porque en caso contrario el flujo del programa seguiría por la primera instrucción del siguiente bloque y así sucesivamente (esto puede ser útil en alguna circunstancia pero no es deseable generalmente). Si variableReferencia no coincide con ninguna de los valores del case, se ejecutará, caso de existir, el bloque correspondiente al *default;* en caso contrario, simplemente se seguiría con la próxima instrucción después del switch.

5.11.2 Bucles

Los bucles son estructuras iterativas que ejecutan un cierto bucle mientras se da una cierta condición. Java dispone de tres tipos de bucles. Tenemos en primer lugar el *while* :

```
while (condicion)
    {bloque}
```

que ejecutará el código del bloque mientras condición se evalúe a cierto. La sentcia *do .. while:*

```
do
    {bloque}
while (condicion)
```

es muy similar sólo que garantiza que el bloque se ejecutará al menos una vez, ya que la evaluación de la condición se produce después de la ejecución del bloque. Finalmente tenemos el clásico *for*:

```
for(inicializazion;condicion;incremento)
    {bloque}
```

que de forma análoga permite ejecutar un bloque mientras se da una cierta condic ión. El *for* tiene de particular que la inicialización (generalmente un contador) y el incremento queda encapsulado en la misma estructura. Es útil para recorrer estructuras de datos secuenciales.

La palabra reservada *break* en cualquiera de los tres bucles fuerza la salida del bucle pasándose a ejecutar la siguiente instrucción después del mismo.

La sentencia *continue* fuerza el abandono de la iteración actual haciendo que se ejecute la siguiente; también para todos los tipos de bucle.

5.12 Aserciones

Las aserciones son una nueva facilidad introducida en la versión 1.4 de Java. Consiste en introducir una serie de predicados lógicos en el código que representan el supuesto estado correcto del programa en ese momento. Si estos predicados no se evalúan a cierto, entonces el programa lanza un error.

Programar usando aserciones incrementa la confianza en que el programa en efecto hace lo que se supone que tiene que hacer además de ser una manera eficaz y sencilla de encontrar y corregir errores de programación.

La palabra reservada que se usa para crear aserciones es, obviamente, *assert* y puede utilizarse de la siguiente manera:

```
assert Expresión1 [: Expresión2] ;
```

La primera expresión sirve para definir la aserción que queremos propiamente controlar; la segunda expresión, que es opcional, sirve para que en caso de no cumplirse la condición, pasarle un valor concreto al error que se produce y que el programador podrá evaluar y utilizar.

Como hemos dicho, las aserciones son una novedad de la versión 1.4 con lo cual la palabra reservada *assert* no lo era antes; por ello, para no tener problemas de compatibilidad a la hora de compilar habrá que usar un parámetro adicional tal como sigue:

```
> javac –source 1.4 MiClase.java
```

Veamos un primer ejemplo trivial. Supongamos que queremos controlar que el valor de una variable esté dentro de un rango determinado:

```
public class Aserciones
{
      public static void main(String[] args)
      {
            int a=10;

            assert a >=0 && a <= 5;

            System.out.println("Todo ha ido bien, a="+a+", está en el
rango 0-5 esperado");
      }
}
```

Si ahora ejecutáis el programa de la forma habitual con el intérprete de Java parecerá que el entorno de ejecución ha ignorado nuestra aserción, y en efecto, así ha sido. Ello se debe a que aunque la aplicación esté programada con aserciones, su comprobación puede activarse y desactivarse a la hora de ejecutar la misma y por defecto el intérprete tiene el tratamiento de aserciones desactivado. Para activarlo explícitamente tenemos que añadir un nuevo parámetro en el intérprete. Por ejemplo, para ejecutar nuestra aplicación:

```
> java –ea Aserciones
```

que produce el resultado esperado:

```
Exception in thread "main" java.lang.AssertionError
        at Aserciones.main(Aserciones.java:15)
```

El parámetro *-ea* es una abreviatura de *–enableassertions* e igualmente tenemos los parámetros *–da* y *–disableassertions*. Si utilizamos estos parámetros tal como en el ejemplo la activación o desactivación de aserciones se está aplicando a todas las clases que se invoquen durante la ejecución de la aplicación excepto las de sistema. Podemos aplicar la activación y desactivación con un nivel de concreción más elevado:

sin argumentos (como en nuestro ejemplo): activa o desactiva aserciones en todas las clases excepto en las de sistema;

nombreDePaquete… : activa o desactiva aserciones en todas las clases que están dentro del paquete *nombreDePaquete* y sus subpaquetes;

… : activa o desactiva aserciones en el paquete sin nombre del directorio de trabajo;

nombreDeClase: activa o desactiva aserciones en la clase *nombreDeClase*.

Si en este momento no entendéis el significado de algún concepto, como el de paquete, no os preocupéis, se aclarará más adelante. Baste decir por el momento que un paquete es simplemente un conjunto de clases agrupadas.

Veamos algún ejemplo:

```
> java –ea: proyecto.dominio... Proyecto
```

ejecuta la aplicación *Proyecto* y activa el tratamiento de aserciones sólo en las clases contenidas en el paquete *proyecto.dominio* y en todos los subpaquetes que pudiera tener. Si pasamos más de un parámetro de control de aserciones, estas se aplican la una detrás de la otra en el orden de escritura:

```
> java –ea: proyecto.dominio... –da:Persona Proyecto
```

imaginaos que la clase *Persona* pertenece al paquete *proyecto.dominio*; con la invocación anterior, por tanto, se activaría el tratamiento de aserciones para todas las clases de dicho paquete excepto para la clase *Persona*.

Para activar o desactivar el tratamiento de aserciones en las clases de sistema hay que usar otros parámetros: *-esa* o *–enablesystemassertions* y *–dsa* o *–disablesystemassertions*.

Hasta hemos estado viendo cómo se usan las aserciones en Java, pero no hemos visto cuándo y por qué se usan. Explicar esto en profundidad y detalle requeriría de una asignatura de un curso de programación básica, por lo que sólo vamos a mencionar unas ideas por encima. Como se mencionaba al principio de la sección el uso de aserciones permite crear software más fiable y donde es más fácil encontrar fallos de programación. Veamos qué tipos de aserciones elementales, según su uso, hay.

5.12.1 Invariantes internos

Antes de que existieran los asertos, muchos programadores usaban comentarios para indicar sus suposiciones sobre el comportamiento de un programa. Por ejemplo:

```
if (a < 0) {...}
else if (a > 0) {...}
else // sabemos que a=0
{...}
```

Disponiendo ahora de la facilidad de las aserciones sería más conveniente escribir:

```
if (a < 0) {...}
else if (a > 0) {...}
else
{
      assert a==0: a;
      ...
}
```

5.12.2 Invariantes de control de flujo

La idea fundamental es que el invariante debe ser cierto mientras se está ejecutando una iteración dentro de un bucle. Esto es útil para garantizar que la condición del bucle ha sido programada de forma adecuada para que este haga realmente lo que el programador espera que haga. Imaginad que queremos recorrer un array de enteros de tamaño 3 mediante un for; asumiendo que el primer índice válido es el 0, sabemos que la variable que usamos para recorrer el vector siempre debe ser inferior a 3 dentro del bucle, así pues nuestro aserto debería comprobar que esto es cierto y en caso de programar mal la condición del bucle el programa lanzaría en ejecución un error y detectaríamos así esta situación anómala. Veámoslo:

```
public class Invariante
{
      public static void main(String [] args)
      {
            int[]vector = new int[3];

            for(int i=0;i<=3;i++) // condición errónea
            {
                  assert i<3 : i;

                  vector[i]=i;
                  System.out.println(vector[i]);
            }
      }
}
```

que compilándolo y ejecutándolo con las opciones adecuadas produciría la siguiente salida:

```
> javac -source 1.4 Invariante.java
> java -ea Invariante
0
1
2
Exception in thread "main" java.lang.AssertionError: 3
        at Invariante.main(Invariante.java:9)
```

Otra variente de los invariantes de control de flujo es la de garantizar que el mismo no llegue a algún punto en concreto. Por ejemplo:

```
void foo()
{
      for (...)
      {
            if (...)
                return;
      }
      // la ejecución nunca debería pasar por aquí!!!
}
```

que con un tratamiento de aserciones adecuado se rescribiría así:

```
void foo()
{
       for (...)
       {
              if (...)
                  return;
       }
       assert false;
}
```

5.12.3 Precondiciones, postcondiciones e invariantes de clase

Las precondiciones son condiciones que como su propio nombre indica deben cumplir los parámetros de una operación a la hora de ser llamada, no obstante su uso es inadecuado en las operaciones públicas porque por convención estas están diseñadas para lanzar excepciones concretas cuando los parámetros no son válidos que ayudan a determinar el porqué (una comprobación mediante asertos sólo sería capaz de lanzar una excepción de aserto), no obstante sí pueden usarse en las operaciones privadas.

Las postcondiciones son predicados lógicos que garantizan que al final de la operación el valor de retorno de los resultados cumple ciertas condiciones, su uso es útil para detectar errores en la implementación de la operación.

Finalmente los invariantes de clase son condiciones que se aplican sobre todas las instancias de una misma clase. Cuando veamos orientación a objetos este último punto quedará más claro.

5.13 El método *main*

El método main es la primera operación que se ejecuta en un programa Java, el que se encarga de poner todo en marcha, y sólo puede haber uno. Su declaración es como sigue:

```
public static void main(String[]args)
```

y siempre debe ser exactamente así, ya que si modificamos (u olvidamos) alguno de los modificadores Java no lo interpreta como el método *main* y cuando intentáramos ejecutar la aplicación obtendríamos un error que precisamente nos diría esto más o menos: “no puedo ejecutar el programa porque no existe un método main”.

Como veis, el método main siempre recibe un parámetro que es un array de String. Este array contiene los parámetros que opcionalmente le hemos podido pasar por la línea de comandos. ¿Qué significa esto? Habitualmente cuando ejecutamos un programa lo hacemos de la siguiente manera:

```
> java HolaMundo
```

pero también habríamos podido ejecutarlo pasándole al programa información adicional:

```
> java HolaMundo p1 p2 p3
```

en este caso hemos llamado a nuestra aplicación y le hemos pasado tres parámetros, p1, p2 y p3, que se almacenan precisamente en el parámetro args del método main. Para acceder a ellos nada más fácil que hacer:

```
public static void main(String[]args)
{
      ...
      String s = args[0];
      ...
}
```

Tened en cuenta que args es un array de String's por lo que aunque nosotros pasemos desde la línea de parámetros números, estos son interpretados como String y si queremos el int, por ejemplo, correspondiente a esa representación tendremos que aplicar los métodos conversores adecuados.

El número de parámetros que se han pasado por la línea de comandos puede consultarse mirando la longitud del array:

```
int numParam = args.length;
```

Ejercicio 5.1

Haced una clase *CuentaAes* cuyo método *main* vaya leyendo caracteres de la entrada estándar hasta que encuentre el carácter '.' y entonces saque por pantalla el número de 'a'.

Para leer un carácter desde la entrada estándar (el teclado) podéis utilizar el comando:

char c= (char) System.in.read();

Para escribir por la salida estándar (pantalla) usad la sentencia:

System.out.println("mensaje")

La cabecera del método main debe capturar las posibles excepciones que se pueden producir debido al manjero de los streams de entrada/salida (todo quedará claro en próximas lecciones):

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

Ejercicio 5.2

Haced una clase *EuroConversor* que transforme valores de pesetas a euros y viceversa. Hay que declarar una "constante" con las pesetas que vale un euro (recordad que las constantes los son de clase por lo que debe estar declarada fuera del método main). Haced que el programa pida primero qué tipo de

conversión quiere hacerse (si de euros a pesetas o de pesetas a euros) y que a continuación muestre el resultado.

Se pueden utilizar los métodos de la clase *Console*:

String readLine(String prompt)
double readDouble(String prompt)
int readInt(String prompt)

Aseguraos de tener la clase *Console* en vuestro directorio de trabajo y para utilizar los métodos debéis hacerlo de la siguiente manera:

double d=readDouble("texto que aparecerá por la consola\r\n");

Ejercicio 5.3

Con ayuda de la clase *Console,* haced una clase *VisorUnicode* que muestre los caracteres Unicode comprendidos entre dos enteros que introducirá el usuario. Deberéis comprobar que los enteros estén dentro del rango de los códigos Unicode (recordad que uno de estos caracteres tiene un tamaño de 2 bytes). Al asignar a un carácter un entero se le está asignando su código Unicode aunque es posible que tengáis que hacer algo para que el compilador os permita esta asignación.

Si queréis hacer pruebas sabed que los dígitos están en los caracteres decimales del 48 al 57 y las letras están del 65 al 90 en mayúsculas y del 97 al 122 en minúsculas.

Ejercicio 5.4

Haced una clase *Palindromo* que compruebe si un *String* es una palabra capicúa (palíndromo). No hace falta que gestionéis la entrada/salida (podéis poner el *String* dentro del mismo código sin preguntarle al usuario).

Ejercicio 5.5

Modificar los ejercicios 5.3 y 5.4 para que los valores se pasen a través de la línea de comandos. Para convertir un *String* que representa un *int* en el int correspondiente se hace así:

int i = Integer.parseInt("3")

Ejercicio 5.6

Haced un programa *Parametros* que pueda recibir un número indeterminado de parámetros y que compruebe que no hay ninguno que sea igual al primero.

6 PROGRAMACIÓN ORIENTADA A OBJETOS

La orientación a objetos es un método de construcción de sistemas software que incluye todas las fases de desarrollo, desde la especificación hasta la programación pasando por el diseño. Actualmente es el paradigma de programación dominante. Java es un lenguaje totalmente orientado a objetos, de forma que resulta imprescindible entender las nociones básicas en que se fundamenta este paradigma para poder hacer una aplicación mínimamente interesante.

Muchas de las ideas están cogidas directamente de la programación con Tipos Abstractos de Datos (TAD's) o son una evolución de las mismas. En este capítulo se describen, sin entrar en mucho detalle, los conceptos clave de la programación orientada a objetos y su materialización en construcciones del lenguaje Java.

6.1 Conceptos

Al construir una aplicación nos fijamos en las entidades del mundo real sobre las que va a tener que realizar cálculos y manipulaciones, lo que se llama comúnmente el **dominio** de la aplicación. Tomemos como ejemplo un sistema para realizar las nóminas de una empresa. Las entidades relevantes serían cada uno de los empleados de dicha empresa, sus horarios, sus cuentas bancarias, etcétera.

Las entidades del mundo real tienen dos características que las definen: un **estado** y un **comportamiento**. Las entidades de tipo empleado tienen un estado (su nombre, su horario, ...) y un comportamiento (cambiar de puesto en la empresa, cambiar de horario, darse de baja...).

En orientación a objetos cada entidad del mundo real relevante para la aplicación tiene su objeto software equivalente. En los objetos software el estado del mismo se representa con uno o más **atributos** (piezas de información identificadas con un nombre). El comportamiento de los objetos software se implementa con **métodos** (también llamados funciones o rutinas) que se aplican al objeto y que pueden manipular el estado de estas variables. Ambos, atributos y métodos, reciben también el nombre de **miembros**.

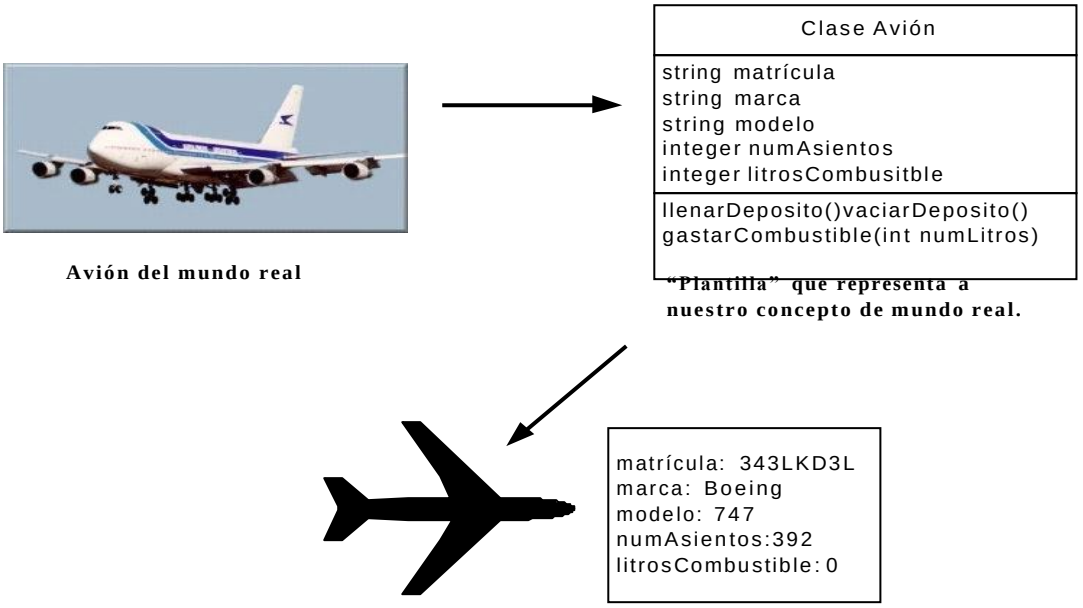
En orientación a objetos un concepto del mundo real se traduce a nivel de código en una **clase**. Cada **objeto** software es una instancia de una clase. Desde este momento hablaremos indistintamente de objeto y de instancia. La diferencia es importante, la clase podría definirse como una plantilla de lo que será el objeto, es decir la clase define que una persona tiene nombre, apellido, fecha de nacimiento etcétera, y un objeto es una persona concreta, es decir un objeto podría ser Sebastián González que nació el 4 de agosto de 1975.

Una clase es un modelo de lo que representa en el mundo real y, como todo modelo, no es una copia exacta del mismo, es una **abstracción**, es decir recoge sólo aquellos detalles, aquella información, que es relevante para la aplicación que se está desarrollando. En efecto si nos fijamos en la figura 5 todos estaremos de acuerdo que un avión no se reduce sólo a su matrícula, marca, modelo, número de asientos y capacidad de su depósito,

sin embargo es perfectamente factible que esta información sea suficiente para algún tipo de aplicación

Una clase define qué variables (llamadas también, como hemos visto, atributos) poseen los objetos de esa clase, y qué métodos se les puede aplicar. Constituyen una especie de manual de construcción de objetos de un tipo concreto. Una vez hemos definido una clase podemos instanciar cuantos objetos de esa clase queramos, lo que equivaldrá a reservar espacio en memoria para estos objetos y sus atributos. La noción de objeto sólo tiene sentido en tiempo de ejecución.

Fig. 5
Concepto, Clase y Objeto



Si vemos las clases como tipos de datos, los métodos que definen son las operaciones aplicables al tipo. Invocar un método sobre un objeto es ejecutar una operación de la clase del objeto, con el propio objeto como parámetro implícito. También se llama a esto pasar un mensaje al objeto.

El concepto de clase es equivalente en muchos aspectos al de TAD. Se trata de encapsular en una misma unidad o módulo los datos y las operaciones que acceden a esos datos. Además los datos no son accesibles directamente sino que hay que usar la interfaz que define el módulo. Esto contribuye a hacer el código más fácilmente modificable puesto que cambios en la representación sólo afectan a las operaciones del propio módulo, y no se propagan al resto de la aplicación. Se reutilizan estructuras de datos completas en lugar de simples rutinas.

La clase tiene una doble vertiente: como tipo de dato que permite crear instancias de ese tipo (objetos) y cómo unidad modular básica que nos ayuda a estructurar el código. En Java y en otros lenguajes orientados a objetos las clases constituyen la única forma de crear nuevos tipos de datos.

6.2 Uso de clases ya existentes

Exceptuando los tipos primitivos explicados en el capítulo anterior, todos los tipos de datos se definen mediante clases. Veremos en este apartado cómo instanciar objetos de una clase e invocar métodos sobre los mismos.

6.2.1 La API de Java

Uno de los atractivos y la gran utilidad de Java radica en la interfaz para programación de aplicaciones (API) de Java. La API consiste en un conjunto de paquetes (packages) que se distribuyen con el J2SDK como bibliotecas de clase. Estos paquetes proporcionan una interfaz común para desarrollar programas en todas las plataformas Java. La API ofrece todas las posibilidades para desarrollar desde programas de consola hasta aplicaciones con interfaz GUI pasando por aplicaciones cliente/servidor, videojuegos entre otros.

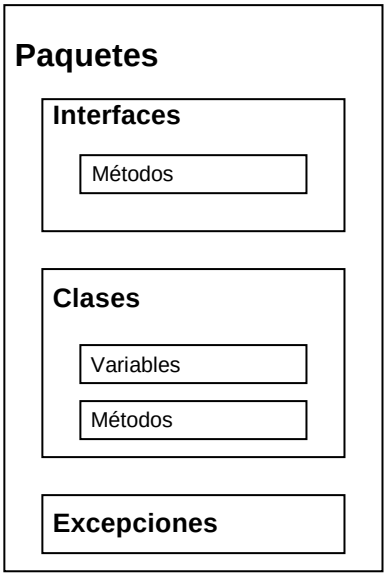


Fig. 6
Organización de los paquetes Java

Los paquetes son conjuntos de clases, interfaces y excepciones (estos dos últimos conceptos se explican más adelante) relacionados. Por ejemplo, vienen separados según se trate de programas de ventana, applets, software de conexión de red, etcétera.

La API está perfectamente documentada en formato html. Esta documentación, como ya se mencionó, hay que descargarla a parte del J2SDK o puede consultarse on-line en la web de Sun en la siguie nte URL: http://java.sun.com/j2se/1.4/docs/api/index.html. Si se descarga, suele copiarse en (suponiendo que el J2SDK se haya instalado en C:\Java) C:\Java\docs\api.

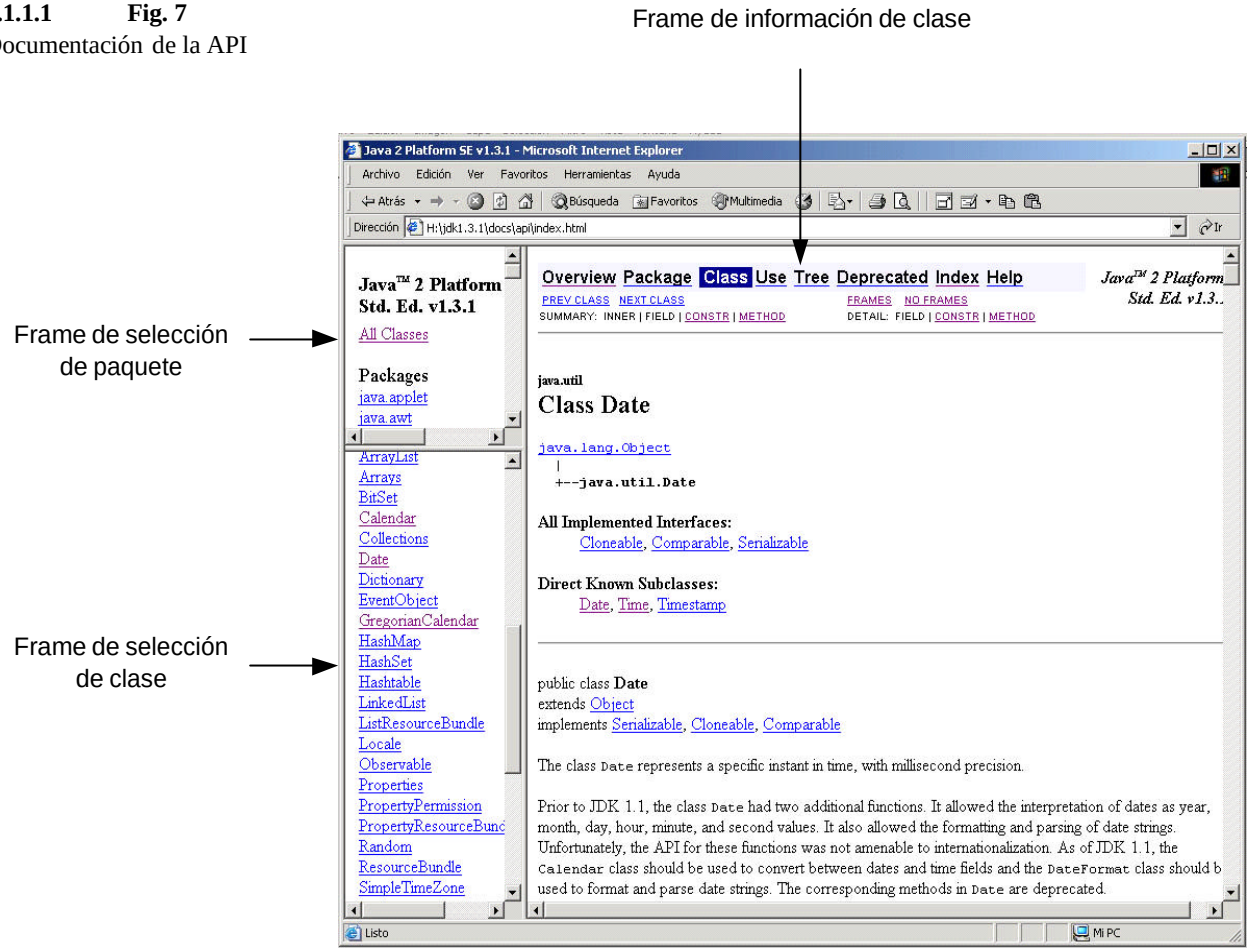
En cualquier caso, al abrirlo con nuestro navegador encontraremos una página dividida en tres frames (figura 7). El frame superior izquierdo permite seleccionar los paquetes de los cuales van a visualizarse clases, el frame izquierdo muestra todas las clases del paquete seleccionado y el frame derecho, el principal, muestra toda la información de la clase que se está consultando.

Veamos cada frame con más detalle. El frame de selección de paquete nos muestra la versión de Java así como la posibilidad de seleccionar todas las clases (figura 8).

Si nos fijamos bien en el frame de selección de clase (figura 9) veremos que algunas de ellas aparecen en cursiva, ello se debe a que son interfaces (lo estudiaremos más adelante).

El frame de información de clase nos aporta toda la información que vamos a necesitar para trabajar con ella: el paquete al que pertenece, el nombre de la clase (o interfaz), el árbol de jerárquico de clases, los interfaces que implementa, una descripción detallada y a continuación se muestran los atributos, los métodos constructores y todos los métodos que tiene la clase.

1.1.1.1 Fig. 7
Documentación de la API



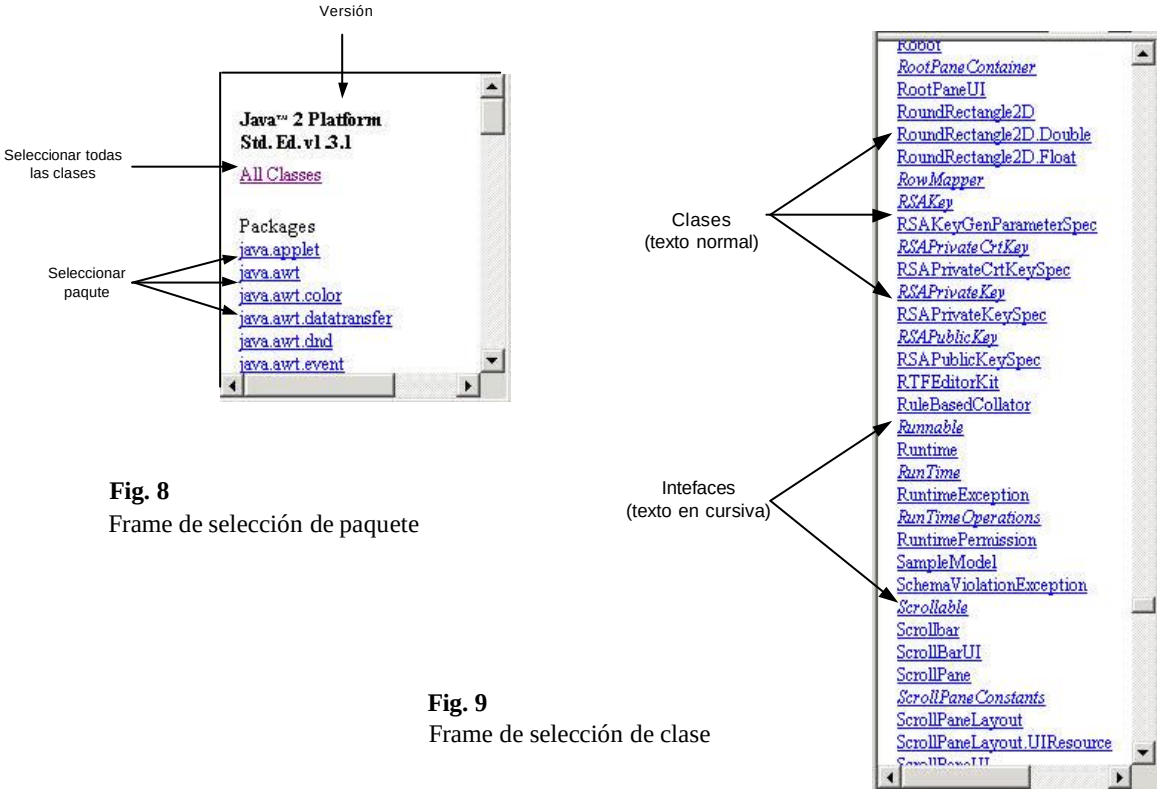


Fig. 8
Frame de selección de paquete

Fig. 9
Frame de selección de clase

6.2.2 Declaración de variables e instanciación de objetos

Como ocurría con los tipos primitivos, para declarar una variable de un tipo definido mediante una clase, hay que escribir el nombre del tipo seguido del identificador (o lista de identificadores). Por ejemplo, existe una clase de librería del J2SE para representar fechas llamado `Date`. Podemos declarar una variable del tipo `Date` con la siguiente instrucción:

```
Date fecha; // 'fecha' no referencia ningún objeto
```

Como ya sabemos, un objeto es una zona de memoria donde se guardan unos datos con una cierta estructura que se ha definido en la clase del objeto. Las variables para tipos no primitivos no son objetos, sino referencias a objetos. Desde este momento hablaremos de variables y referencias a objeto indistintamente. En realidad la variable `fecha` no referencia todavía ningún objeto. Como los métodos se ejecutan sobre algún objeto, aplicar un método sobre esta variable producirá un error de ejecución.

Toda variable de tipo no primitivo es inicializada a *`null`* y no apunta a ningún objeto hasta que le damos valor explícitamente mediante una asignación. También es posible desreferenciar una variable asignándole *`null`*. Para crear un objeto al cuál poder referenciar debemos utilizar el operador *`new`:*

```
fecha = new Date(); // ahora hemos creado una instancia
```

Lo que aparece detrás del *`new`* es un constructor. Un constructor es un tipo especial de operación que inicializa un objeto recién creado. Cada clase tiene uno o varios constructores cuyo nombre debe coincidir con el de la clase. Los constructores pueden tener parámetros. El constructor del ejemplo inicializa el nuevo objeto de tal modo que representa la fecha y hora actual.

La clase tiene otro constructor que nos permite especificar una fecha concreta mediante un parámetro que representa el número de milisegundos desde las 00:00h del 1 de enero de 1970:

```
Date fecha2 = new Date(123412123412341341);
```

Vemos aquí que podemos inicializar una variable (es decir, asignarle algún objeto al cuál referenciar) en el mismo momento que la declaramos.

6.2.3 Invocación de métodos y acceso a atributos

Cuando una variable referencia algún objeto podemos usarla para invocar métodos sobre ese objeto. La interfaz del objeto (los métodos que es posible invocar sobre el mismo) estará definida en la clase a la que pertenece el objeto. La clase `Date` ofrece una serie de métodos para manipular fechas. Tiene definido por ejemplo un método `after(Date when)` que devuelve un `boolean` con valor `true` si el objeto de tipo `Date` sobre el que invocamos el método representa una fecha posterior al parámetro `when`. La sintaxis para invocar un método sobre un objeto es

```
referenciaAObjeto.nombreMéodo(lista_parámetros)
```

por ejemplo:

```
boolean isAfter = date1.after(date2);
```

De modo análogo podemos acceder a los atributos de un objeto con la sintaxis:

```
referenciaAObjeto.nombreAtributo
```

Accedemos de igual forma a métodos y atributos con la diferencia que en los métodos hay que escribir la lista de parámetros entre paréntesis. Es importante remarcar que aunque no tenga parámetros, deberá llevar los paréntesis , ya que en caso contrario asumirá que estamos intentando acceder a un atributo.

6.3 Creación de nuestras propias clases

Vamos a ver ahora cómo definir un nuevo tipo de datos mediante la definición de una clase. En una clase se declaran tanto los datos que van a constituir la representación interna de los objetos de esa clase (atributos) como las operaciones que accederán a esos datos (métodos). Métodos y atributos constituyen los miembros de la clase

6.3.1 Cuerpo de la clase

Modelaremos como ejemplo el concepto de empleado. Para cada empleado se guarda un nombre, un sueldo y una fecha de contratación. Sobre un objeto de tipo empleado podemos invocar métodos para conocer su nombre, incrementar su sueldo en un porcentaje y obtener el año en que fue contratado:

```
01    import java.util.Calendar;
02    import java.util.GregorianCalendar;
03
04    class Empleado
05    {
06          private String nombre;
07          private double salario;
08          private GregorianCalendar fechaContratacion;
09
10          public Empleado(String n, double s, GregorianCalendar d)
11          {
12                nombre = n;
13                salario = s;
14                fechaContratacion = d;
15          }
16
17          public String getNombre()
18          {
19                return nombre;
20          }
21
22          public void setNombre(String nuevoNombre)
23          {
24                nombre = nuevoNombre;
25          }
26
27          public double getSalario()
28          {
29                return salario;
30          }
31
```

```
32          public void aumentarSalario(double porPorcentaje)
33          {
34                salario *= 1 + porPorcentaje / 100;
35          }
36
37          public GregorianCalendar getFechaContratacion()
38          {
39                return fechaContratacion;
40          }
41
42          public int añoContratacion()
43          {
44                return fechaContratacion.get(Calendar.YEAR);
45          }
41    }
```

En las dos primeras líneas de código se importa dos clases para su uso posterior. Se hablará de esto más adelante.

Justo detrás de la palabra reservada *class* escribimos el nombre de la clase, y a continuación y entre llaves, la lista de atributos y métodos. La clase que nos sirve de ejemplo tiene tres atributos llamados *nombre*, *salario* y *fechaContratacion*. Podemos ver que los atributos pueden ser de tipo primitivo o bien referencias a otros objetos.

La clase tiene también seis métodos. Cuatro de ellos (*getNombre()*, *getSalario()*, *getFechaContratacion* y *añoContratacion()*) son consultores, puesto que informan del estado del objeto sobre el cual se invocan, pero no modifican tal estado. Los otros dos (*setNombre(...)* y *aumentarSalario(...)*) son modificadores puesto que cambian el valor de algún atributo y por tanto el estado del objeto.

Los métodos incluyen su cabecera y su implementación. Los atributos se declaran como cualquier otra variable, pero son externos a cualquier método. El orden en que aparecen atributos y métodos en el cuerpo de la clase no tiene ninguna importancia. No hay que declarar cabeceras cómo en C/C++.

Los métodos tienen que declarar su valor de retorno (`void` en caso que no devuelvan nada). A continuación del nombre y separados por comas se escriben los parámetros formales.

En Java hay sobrecarga de operaciones (*overloading*). Esto significa que podemos declarar en una misma clase dos o más operaciones con el mismo nombre. Para que tal cosa sea posible dichas operaciones tienen que diferenciarse en el número o tipo de sus parámetros.

La clase y sus atributos y métodos pueden llevar unos modificadores que afectan su funcionamiento: *public*, *private*, *protected*, *static*, *synchronized* y *final*. Iremos viendo el significado de la mayoría de ellos a lo largo del capítulo.

6.3.2 Constructores

Toda clase tiene que tener al menos un constructor, así que si no declaramos ninguno dispondremos por lo menos del llamado constructor por defecto o de oficio. Dicho constructor no tiene parámetros y da valor 0 (numéricos), `false` (booleano), '\u0000' (carácter) o `null` a todos los atributos no inicializados explícitamente en su declaración.

En este caso la clase *`Empleado`* no dispone de constructor sin parámetros porque en las líneas 10 a 15 se define uno con tres parámetros que inicializa todos los atributos del objeto creado. Es recomendable que esto ocurra para evitar errores por acceso a atributos con valor `null`.

Podríamos haber definido otros constructores siempre que hubieran tenido un número o tipo diferente de parámetros. Es muy habitual el uso de la sobrecarga en los constructores.

Los constructores no tienen valor de retorno (no hay que especificar `void`)

> **Ejercicio 6.1**
>
> Crear una clase *PruebaEmpleado* (no hace falta gestionar la e/s) para probar la clase *Empleado*. Se tienen que instanciar unos cuantos empleados, mostrarlos por pantalla, aumentarles el sueldo en un porcentaje y finalmente mostrar de nuevo los datos para comprobar que se han modificado. Os puede ser de utilidad introducir los empleados en un array que podéis recorrer para imprimir los datos.

6.3.3 Acceso a los miembros del propio objeto

Los métodos vistos hasta el momento se invocan sobre un objeto. En el código de un método podemos acceder a los atributos del objeto con solo escribir el nombre del atributo. Tenemos un ejemplo de ello en la línea 19 de *`Empleado`* con

```
return name;
```

No obstante hay una forma de explicitar que se está referenciando el objeto que recibe el mensaje con *`this`*. Se podría haber escrito

```
return this.name;
```

Del mismo modo se pueden hacer llamadas a métodos sobre el mismo objeto con sólo escribir su nombre y parámetros.

.

6.3.4 Visibilidad **public** *y* **private**

El diseñador de una clase debe asegurarse que la interacción con los objetos de la clase se realiza a través de un interfaz simple y bien definida. Este interfaz no debe permitir hacer un uso inadecuado de la clase. Imaginemos por ejemplo, que tenemos una clase *Pila* implementada con un *array* y dos enteros para la posición de la cima y el número de elementos apilados. Sería muy perjudicial que un usuario incauto de la clase *`Pila`* pudiera

modificar los valores de los dos enteros a su antojo porque podría obtener instancias inconsistentes del tipo `Pila`. Es evidente que dichos enteros deberían modificarse sólo dentro de la clase `Pila`

Las verdaderas ventajas de la encapsulación surgen cuando usamos también la ocultación de información, esto es, que ciertas propiedades de los objetos de la clase no sean accesibles para el código externo a la misma.

En Java podemos declarar los miembros como `private`, haciéndolos visibles únicamente al código de la clase, o cómo `public`, siendo entonces accesibles desde cualquier otra clase. Los atributos `public` son accesibles desde el exterior tanto para consulta como para modificación. Existen otras categorías de visibilidad que ya comentaremos.

Por lo general los atributos deben hacerse siempre `private` (exceptuando las constantes de clase de las que ya hablaremos). Es muy peligroso permitir acceder a la representación interna de un tipo sin usar el interfaz previsto para tal fin. En el caso de los métodos, una clase tiene que ofrecer algún método `public` para ser de utilidad. Es posible que para implementar los métodos `public` hagamos uso de diseño descendente y queramos definir operaciones que no se vayan a usar fuera de la clase, podemos entonces declararlas `private`.

En la clase `Empleado` se han seguido estos criterios. Todos los atributos son `private`, y si para alguno de ellos resulta importante poder conocer o modificar su valor se han proporcionado métodos consultores y modificadores (*get* y *set)* apropiados.

Los constructores son siempre `public`. y no devuelven ningún valor.

6.3.5 Paso de parámetros

Todos los parámetros a métodos se pasan por valor. Esto resulta claro en los tipos primitivos pero se debe refinar la idea en el caso de pasar objetos como parámetros.

El paso por valor supone que el método invocado tiene su propia referencia al objeto que se pasa como parámetro, distinta de la del método invocador, y quiero enfatizar que lo que se **copia** es la **referencia** y **no** el **objeto**. Esto implica que si el método invocado realiza asignaciones sobre su parámetro formal (una copia de la referencia del invocador), estas no afectaran a la referencia que el método invocador tenía al objeto: seguirán apuntando al mismo objeto, pero por otro lado, el método invocado puede usar su referencia al objeto para modificar su estado (por ejemplo usando métodos modificadores). Esto sugiere que los métodos pueden provocar efectos colaterales sobre los objetos que pasamos como parámetros. Por ello, si queremos garantizar que nuestro objeto no va a ser modificado dentro del código del método invocado deberemos pasarle la referencia a un objeto copia del que queremos proteger, de ahí la necesidad del uso de métodos *clone()*. Todo esto quedará más claro en el apartado siguiente.

6.3.6 Asignaciones, copias y comparaciones

Cuando realizamos una asignación de tipos no primitivos, en realidad estamos asignando punteros, obteniendo dos referencias a un mismo objeto. Veamos un ejemplo de lo que esto puede comportar:

```
Empleado a = new Empleado("David", 2000, new GregorianCalendar());
Empleado b = a;

String str = a.getName();     // str tiene valor "Raúl"
```

Cuando usamos el operador de comparación `==` con dos variables de tipo no primitivo, estamos comparando punteros. Esto quiere decir que tal comparación sólo nos devolverá `true` si ambas variables son referencias al mismo objeto.

A veces nos gustaría poder comparar dos objetos distintos en memoria y saber si tienen el mismo valor para cada uno de sus atributos. Existe un modo elegante de conseguir esto en Java y es dotando a la clase cuyos obje tos nos interese comparar con un método *`equals(...)`* que defina la comparación de un modo razonable para esa clase. Es lo que hace, como ya hemos visto, la clase `String`. También nos puede interesar replicar un objeto de forma que existan dos copias del mismo en memoria. Habrá que escribir un método *`clone()`* que cree un nuevo objeto y inicialice sus atributos con los mismos valores que el objeto original. Más adelante daremos algún detalle más sobre este tema.

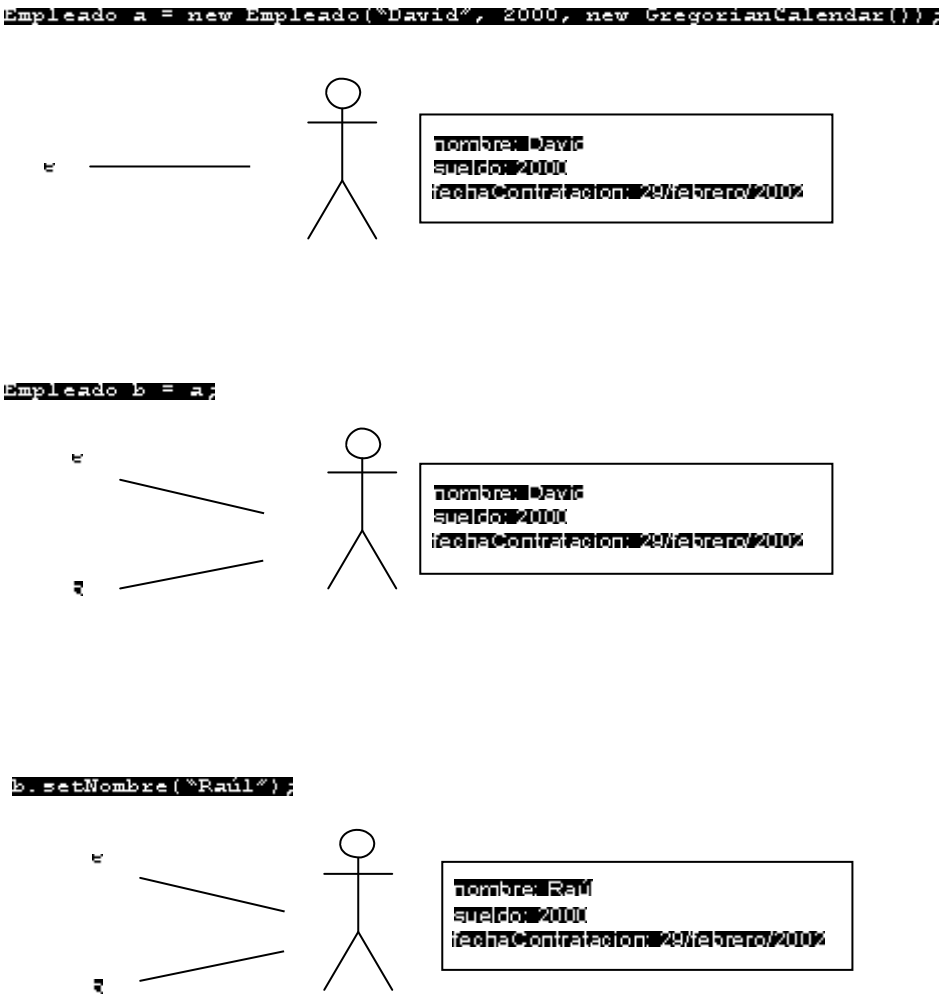


Fig. 10
Problemas de la asignación de tipos no primitivos

6.3.7 Destruir objetos

En la mayoría de lenguajes tenemos que preocuparnos de liberar adecuadamente toda la memoria que vamos reservando dinámicamente a lo largo de una ejecución. Para tal fin se suele dotar a cada TAD con un método destructor que llamamos cuando estamos seguros que no vamos a acceder más a un objeto determinado. El método destructor debe advertir al sistema operativo que la zona de memoria ocupada por el objeto queda libre y puede ser por tanto utilizada para colocar otros objetos.

Todo esto no es necesario en Java porque el entorno de ejecución lleva la cuenta de las referencias que en cada momento apuntan a un objeto determinado. Cuando un objeto pierde la última referencia no vamos a poder acceder jamás al mismo. Este hecho es detectado por el entorno de ejecución que marca ese objeto como eliminable. Cada cierto tiempo la memoria ocupada por los objetos eliminables es liberada automáticamente por la máquina virtual. Esto es lo que se llama recolección automática de basura (*automatic garbage collector*) y ahorra tener que preocuparnos de liberar la memoria que vamos reservando dinámicamente, evitando muchos errores de programación así como el consumo inútil de memoria.

A veces hay objetos que utilizan otros recursos que no son memoria. Por ejemplo un objeto puede tener abierto un canal de acceso a un fichero. Si no cerramos explícitamente el canal, el fichero permanecerá abierto. Sería razonable cerrar el fichero cuando el objeto que puede acceder al mismo resulta liberado. Puesto que no vamos a poder usar más el fichero, mantenerlo abierto es un consumo inútil de recursos.

Para estos casos Java proporciona la posibilidad de definir en cualquier clase un método `finalize()` que se ejecutará cuando el objeto sea reclamado por el recolector automático de basura. Este método puede liberar los recursos que usaba el objeto y que no se van a poder utilizar más. Hay que remarcar que no controlamos en qué momento se va a destruir un objeto, y no sabemos por tanto cuándo el sistema de ejecución va a llamar a `finalize()`, con lo que si la liberación de recursos del tipo comentado es urgente deberemos realizarla manualmente mediante algún otro método.

6.3.8 Miembros de clase

A veces conviene declarar atributos que no pertenecen a ninguna instancia en particular de una clase. O métodos cuyo comportamiento no depende de la instancia sobre cuál se invocan. Son los miembros de clase (en contraposición a los atributos y métodos vistos hasta ahora que son miembros de instancia).

Un atributo de clase puede servir, por ejemplo, para contabilizar cuántas instancias de una determinada clase se crean a lo largo de la ejecución de un programa. Tal atributo es una propiedad de toda la clase en su conjunto, y no de ninguna instancia en particular. Para declarar miembros de clase debemos poner la palabra reservada *`static`*, antes de la declaración del tipo y después de la declaración de la visibilidad:

```
public static int numInstancias;
```

Un método `static` será aquél que no opere sobre ninguna instancia. Esto quiere decir que no podrá acceder a ningún atributo o método que no sea a su vez `static`, si no es sobre objetos que se creen en el propio método o bien se le pasen como parámetro.

Los miembros `static` se pueden llamar de la misma forma que los que no lo son, usando una referencia si los accedemos externamente a la clase a la cual pertenecen:

```
referenciaAObjeto.nombreAtributoStatic
referenciaAOobjeto.nombreMétodoStatic(lista_parámetros)
```

o simplemente por su nombre desde la misma clase.

Pero dado que no acceden en realidad a ningún objeto, podemos llamarlos también usando directamente el nombre de la clase donde están definidos. Vimos ya ejemplos de métodos `static` que son usados de este modo cuando hablamos de la clase *`Math`*:

```
double x = Math.pow(2, 5);    // 2 elevado a 5
```

Esta clase contiene numerosas operaciones aritméticas, trigonométricas, etc. que no se ejecutan sobre ningún objeto, sino que son función exclusivamente de sus parámetros.

La operación

```
public static void main(String[] args)
```

es otro ejemplo de método `static`.

Un uso habitual de los atributos `static` es en conjunción con la palabra reservada `final`. A un atributo final sólo se le puede dar valor (esto es, asignarle algo) una única vez. Esto nos da una forma natural de declarar constantes. Un ejemplo de "constante" es Pi definida en la clase `Math` y accesible mediante

```
Math.PI
```

En Java toda constante está definida en alguna clase.

Ejercicio 6.2

Implementar una clase *Punto2D*:

1. Ha de tener dos constructores, uno con coordenadas cartesianas como constructores (utilizad el tipo double) y otro sin coordenadas que crea el punto (0,0). Dotad a la clase de métodos para trasladar un punto, comprobar la distancia respecto al origen de coordenadas y comprobar la distancia respecto a otro punto (sobrecargando el método distancia). Haced una clase *PruebaPunto* que instancie varios puntos, los traslade y calcule distancias. Para mostrar los datos de un punto se tienen que utilizar los métodos consultores de las coordenadas cartesianas *getX()* y *getY()*. Recordad que la distancia entre dos puntos se calcula como $\sqrt{x_1 \quad x_2^{\ 2} \quad y_1 \quad y_2^{\ 2}}$. Utilizad los métodos de la clase java.lang.Math.

2. Añadid un método *toString()*. Este método que devuelve un string y lo tienen todas las clases Java, sirve para dar una representación en forma de string del objeto sobre el que se invoca y se llama directamente cuando por ejemplo se concatena nuestro objeto a otro String. Haced que dé una representación del estilo (2.5,1.0) Modificad *PruebaPunto* para utilizar este método.

3. Definid adecuadamente el método y *equals* y comprobad que funciona modificando *PruebaPunto*.

Ejercicio 6.3

Modificad la clase *Empleado* de manera que no se guarde el sueldo real sino el sueldo base por hora. Cread una nueva clase *Turno* con los métodos *int getHoraInicio()*, *int getHoraFin()* y *int obtenerHorasPorSemana()*. La hora de inicio y fin del turno se pasan en el constructor. Las horas totales tienen que calcularse suponiendo que se trabaja 5 días por semanal. El *obtenerSalario()* de la clase *Empleado* de calcula en función de las horas totales del turno asignado al empleado correspondiente. Debe ser posible asignar un turno a un empleado en su construcción y deberán añadirse los métodos *getTurno()* y *setTurno()*. Cread una clase *PruebaEmpleadoTurno()* que compruebe que todo funcione bien.

6.4 Herencia

Quizá la aportación más importante de la orientación a objetos respecto a los paradigmas anteriores sea la posibilidad de definir nuevos tipos de datos a partir de los ya existentes mediante el mecanismo de herencia. La herencia consiste en definir una nueva clase describiendo únicamente las características que la diferencian de alguna clase previamente definida.

6.4.1 Conceptos y terminología

Retomemos la clase `Empleado` y supongamos por ejemplo que nuestra aplicación necesita manejar un nuevo concepto, el de directivo. Supongamos que en nuestra empresa los directivos tienen varias características que los diferencian de los empleados normales: tienen una secretaria y sus aumentos de sueldo se calculan de forma diferente al resto de empleados. Esto quiere decir que habrá que definir una nueva clase `Directivo`. Pero hay otras características que los hacen parecidos al resto de empleados: tienen una fecha de contratación, un nombre y un sueldo. Vamos a aprovechar lo que ya hemos programado en `Empleado` a la hora de programar `Directivo` haciendo que `Directivo` herede de `Empleado`:

```
01    import java.util.*;
02
03    class Directivo extends Empleado
```

```
04    {
05          private String nombreSecretaria;
06
07          public Directivo(String n, double s, GregorianCalendar d)
08          {
09                super(n, s, d);
10                nombreSecretaria = "";
11          }
12
13          public void aumentarSalario(double porPorcentaje)
14          {
15                // añadir 1/2% bonus de por cada año de servicio
16                GregorianCalendar today = new GregorianCalendar();
17                double bonus = 0.5 * (today.get(Calendar.YEAR) -
añoContratacion());
18                super.aumentarSalario(porPorcentaje + bonus);
19          }
20
21          public void setNombreSecretaria(String n)
22          {
23                nombreSecretaria = n;
24          }
25
26          public String getNombreSecretaria()
27          {
28                return nombreSecretaria;
29          }
30    }
```

En este caso decimos que *Empleado* es la superclase de *Directivo*, y *Directivo* es una subclase de *Empleado*. La sintaxis general para indicar relaciones de herencia es:

```
class NombreSubclase extends NombreSuperclase...
```

como en nuestro ejemplo se ve en la línea 3.

En Java cada clase tiene una única superclase (lo que llamamos herencia simple), de modo que a la palabra reservada *extends* le sigue un único identificador de clase. La relación de herencia establece una jerarquía entre todas las clases. De todas las clases que heredan de una clase A, directa o indirectamente, decimos que son descendientes de A y para todas ellas A es un antecesor.

Con esto hemos conseguido que *Directivo* disponga en realidad de todos los atributos y métodos que ya poseía *Empleado*, y disponga además de un nuevo atributo *nombreSecretaria* (línea 5) y dos métodos *getNombreSecretaria()* (línea 26) y *setNombreSecretaria(...)* (línea 21).

6.4.2 Sobrescribir métodos

Observamos también en el ejemplo de *Directivo* que el método *aumentarSalario(...)*, que ya se heredaba de *Empleado*, se ha vuelto a definir con un comportamiento distinto. A esto se le llama sobrescribir un método (*overriding*). Para sobrescribir un método su nombre, su tipo, y el número y tipo de los parámetros deben coincidir con los del método sobrescrito. En caso contrario se estará definiendo un nuevo método (recordemos que hay sobrecarga de métodos).

Es posible llamar al método que resulta sobrescrito (en este caso el `aumentarSalario(...)` de `Empleado`) con la palabra reservada *super*:

```
super.metodoSobrescrito(parámetros);
```

Sobrescribir métodos sirve para refinar el comportamiento en la subclase o calcular lo que ya se calculaba en la superclase de un modo alternativo, pero hay que mantener el significado de lo que se está calculando. No deberíamos usar el método `aumentarSalario(...)` para calcular cosas totalmente diferentes a un aumento de sueldo.

Los constructores no se heredan. Recordemos que si no declaramos ningún constructor Java utilizará un constructor por defecto que inicializa todos los atributos definidos en la clase a cero o `null`. La clase `Directivo` define su propio constructor. Lo que hace en la línea 9 con *super* es llamar al constructor de la superclase. Esto se puede hacer siempre que se trate de la primera línea del constructor. De hecho, si no lo hacemos Java lo hará por nosotros, pero llamará siempre al constructor sin parámetros (que puede no existir, cómo ocurre en este caso con `Empleado`).

6.4.3 La herencia como especialización

La relación de herencia es de especialización hacia las subclases y de generalización en sentido opuesto. De alguna forma todo directivo es también un empleado (es un caso particular de empleado, con todas las características fundamentales de empleado y alguna característica extra). Con la herencia sólo podemos aumentar o concretar la funcionalidad que ya teníamos en la superclase: esto es, básicamente, añadir atributos y métodos, y cambiar la implementación de algunos métodos (pero no su interfaz). No podemos eliminar métodos o atributos.

Pero la herencia no es sólo una forma de escribir menos código. Quizá la propiedad más destacable de la herencia es que allí donde esperamos un objeto de una clase, podemos utilizar un objeto de cualquier clase que descienda de la misma. Por ejemplo, a un método que espere un parámetro de tipo *Empleado* podemos llamarlo pasándole un *Manager*. Veremos más aplicaciones de este principio cuando hablemos de polimorfismo.

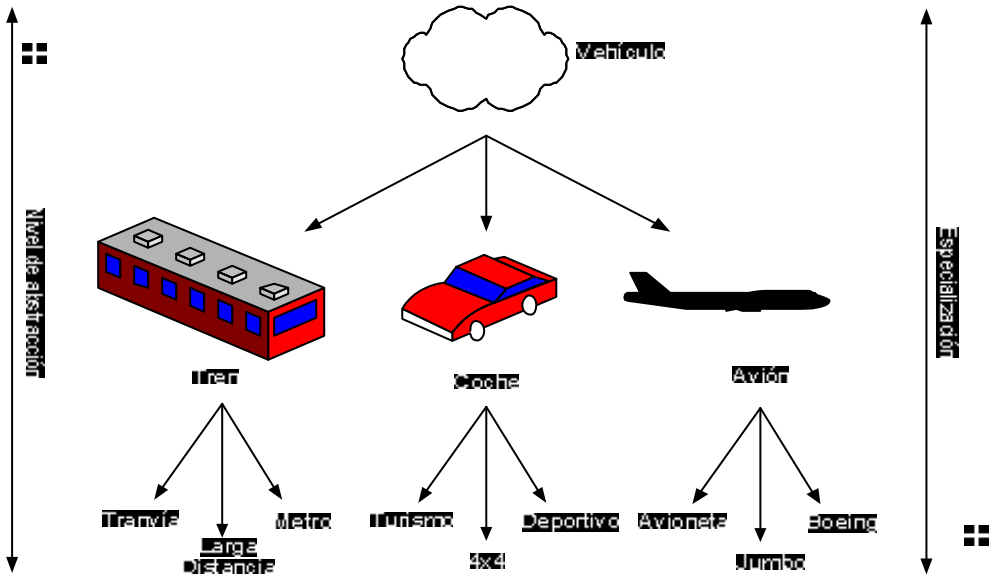


Fig. 11
La herencia como especialización

6.4.4 Visibilidad protected

Un objeto de la clase `Directivo` tiene cuatro atributos: `nombre`, `salario`, `fechaContratacion` y `nombreSecretaria`. Pero los tres primeros están declarados en `Empleado` con visibilidad `private`. Esto significa que no podemos acceder a ellos desde el código de la clase `Directivo`. Sólo podremos modificar o consultar los valores de estos atributos indirectamente con llamadas a métodos `public` de la clase `Empleado`. Así pues si quisiéramos obtener el nombre del empleado deberíamos hacerlos a través del consultor *getNombre:*

```
// Código que habría en la clase Directivo

String n = Nombre // Error! Atributo privado en Empeleado
String n = getNombre // Correcto! El método es público en Empleado
```

Lo mismo ocurre con los métodos `private`: no se pueden llamar desde las subclases.

Para solventar esto que podría parecer engorroso pero no sin ello renunciar a las propiedades en ocultación de la información y encapsulamiento, hay un tipo de visibilidad en Java que permite el acceso a miembros a todos los descendientes de la clase, pero no a todas las clases en general: la visibilidad *protected*.

6.4.5 Polimorfismo y dynamic binding

El polimorfismo es una característica de todos los lenguajes que usan herencia. Consiste en que una misma referencia a objeto puede referenciar en distintas ejecuciones, o en distintos momentos de una misma ejecución, a objetos con diferente tipo (diferente forma). Esto es debido a que a una variable de un tipo A podemos no sólo asignarle objetos del tipo A, sino también cualquier objeto de un tipo que descienda de A.

La siguiente instrucción por tanto es correcta:

```
Empleado jefe = new Directivo(...);
```

El sistema de tipos de Java garantiza que todo miembro que accedamos mediante una referencia determinada estará disponible en la clase del objeto referenciado. Por eso el compilador sólo deja acceder a miembros definidos en la clase de la referencia o en antecesores suyos. Es correcta la llamada

```
jefe.getSalario()
```

pero no

```
jefe.getNombreSecretaria();
// Error de compilación, la clase Employee no dispone de este método
```

Para obtener el nombre de la secretaria del objeto del ejemplo habría que haberlo asignado a una referencia de tipo *Directivo*.

Por otro lado podemos llamar a un método para el cual existan varias implementaciones. En la llamada

```
jefe.aumentarSalario(30);
```

¿Qué implementación se ejecutará, la de `Empleado` o la de `Directivo`? La decisión de qué método utilizar se toma en tiempo de ejecución, y se ejecuta el de la clase a la cual pertenece el objeto referenciado, en este caso la clase `Directivo`. Este modo de proceder se conoce con el nombre de *enlace dinámico* (dynamic binding).

El polimorfismo y el enlace dinámico pueden aprovecharse para hacer tratamientos generales de un conjunto de objetos de forma que para cada objeto se realice el cálculo adecuado sin tenernos que preocupar del tipo específico de cada objeto. El siguiente ejemplo aumenta el salario a tres empleados. A dos de ellos del modo habitual, y al tercero, por ser directivo, le aplica una bonificación por antigüedad:

```
Empleado[] plantilla = new Empleado[3];
plantilla[0] = new Empleado(...);
plantilla[1] = new Empleado(...);
plantilla[2] = new Directivo(...);
for (int i = 0; i < 3; i++) plantilla[i].aumentarSalario(5);
```

Con sólo definir para cada clase su comportamiento exacto mediante sus métodos conseguimos ahorrarnos muchas estructuras condicionales en diferentes partes del código.

6.4.6 Casting *de objetos*

Un objeto no puede cambiar de tipo a lo largo de su vida. No obstante, a veces es necesario usar el *casting* para indicar al compilador que el objeto indicado es en realidad de un tipo más especializado de lo que él puede deducir. Los *castings* están sólo permitidos dentro de una línea de herencia:

```
Empleado jefe = new Directivo(...);
Directivo m = (Directivo) jefe;
```

Sin el *casting* la segunda asignación no seria correcta, porque la variable `jefe` es de tipo `Empleado`, y no todo `Empleado` es un `Directivo`. El *casting* sólo debe hacerse si

estamos seguros que el objeto convertido es del tipo indicado. El siguiente código producirá un error de ejecución, por no referenciar *jefe* a ningún `Manager`:

```
Empleado jefe = new Empleado(...);
Directivo m = (Directivo) jefe;
```

Para evitar problemas a veces es conveniente comprobar si el tipo de un objeto es el adecuado. Podemos hacerlo con el operador *instanceof*, que dado un objeto y un identificador de clase devuelve `true` si y sólo si el objeto pertenece a la clase o a algún descendiente de la clase:

```
Empleado e;
...
if (e instanceof Directivo) { Directivo m =
      (Directivo) e;
      System.out.println(m.getNombreSecretaria());
}
else {
      System.out.println("No  es un Directivo");
}
```

6.4.7 La clase Object

Cuando no especificamos superclase alguna de la cual heredar, estaremos heredando implícitamente de *Object.* Así todas las clases en Java son descendientes de *Object*, lo que permite definir cierto comportamiento común a todas las clases.

Por ejemplo, en la clase *Object* hay un método `equals(Object obj)` (del cuál ya hemos hablado) para comparar el objeto actual con el que se pasa por parámetro. Tal y como se encuentra definido en *Object* el método usa `==` para la comparación, es decir, devuelve *true* sólo si las dos referencias apuntan al mismo objeto en memoria. Pero en cualquier clase que nosotros hagamos podemos sobrescribir este método para que la comparación se comporte adecuadamente. Por ejemplo, podemos hacer que se retorne *true* cuando ambos objetos referenciados coincidan en todos y cada uno de sus atributos. También los métodos *finalize()* y *clone()* (comentados en apartados previos) están definidos en *Object*, en este caso con código vacío.

Hay otro método interesante en *Object* llamado *toString()*. Este devuelve una representación del objeto en forma de String. Podemos sobrescribirlo para devolver una combinación de los atributos del objeto que nos ayude a identificarlo. Este método se llama cuando usamos el operador + para concatenar *Strings* con objetos.

> **Ejercicio 6.4**
>
> Modificar la clase (partiendo de la versión de la documentación) *Empleado* para dotarla de los correspondientes métodos *equals, clone* y *toString* Crear una clase *PruebaEmpleado2* que pruebe estos nuevos métodos

Ejercicio 6.5

Implementar la clase *Directivo* utilizando como guía la documentación (usaremos la versión de *Empleado* del ejercicio anterior).

1. Un *Directivo* tiene como subordinados a un número máximo de *Subordinados.* Este número máximo es una constante de clase. Un *Directivo* debe poder almacenar a sus subordinados; usad un array y cread un método que permita añadirlos (no hace falta el consultor). Hay un subordinado especial, la secretaria: en lugar de los métodos *getNombreSecretaria()* y *setNombreSecretaria(String nombre)*, implementad los métodos *Empleado getSecretaria()* y *setSecretaria(Empleado e)*. Redefinid el método *aumentarSalario* para que se añada un 5% de bonificación por año de antigüedad.

2. Crear una clase *PruebaDirectivo* que contenga un array con unos cuantos empleados, directivos y empleados y hacer un bucle que les suba a todos el sueldo en un mismo porcentaje, pero para cada objeto a su manera (con el plus por bonificación a los directivos), sin usar ningún *if* o *instanceof*. Hacer otro bucle que muestre los nuevos sueldos de todos los empelados.

3. Añadir a un *Directivo* un método privado que obtenga un listado de nombres de todos sus subordinados en forma de string. Usando este método, redefinir el método *toString* de manera que a continuación del nombre y del año de contratación se muestre la lista de subordinados.

4. Hacer un bucle que imprima todos los nombres de los empleados y en el caso de los directivos que imprima al lado el nombre de la secretaria. Usad *instanceof.*

6.4.8 Clases abstractas

A medida que subimos en la jerarquía de herencia nos encontramos con clases que representan conceptos cada vez más abstractos. Puede tener sentido modelar conceptos abstractos de los cuales no queremos crear instancias directamente, pero que nos son útiles para definir propiedades compartidas por varias subclases. Estas clases pueden ser tan generales que no sea posible dar una implementación para alguno de sus métodos, dejando entonces tal tarea para las subclases.

Tomemos como ejemplo una aplicación donde hay que dibujar una serie de objetos gráficos como círculos, rectángulos, líneas, curvas de Bezier, etc. Para cada tipo de objeto gráfico definiremos una clase. Hay ciertas propiedades comunes a todas estas clases de objeto: todo objeto gráfico tendrá unas coordenadas de referencia y queremos que se pueda mover de sitio. Por eso definimos una clase *ObjetoGrafico* de la que heredarán el resto:

```
abstract class ObjetoGrafico
{
      int x, y;
      ...
      public void moverA(int newX, int newY)
{
      ...
      }

      public abstract void dibujar();
}
```

Algunos métodos como *moveA(...)* podrán ser implementados en esta clase. Pero, ¿qué ocurre con *dibujar()*? Cada tipo específico de objeto gráfico va a dibujarse de forma distinta y resulta imposible dar una implementación suficientemente general para todos. Lo que se hace en el ejemplo es declarar el método como *abstract*, indicando que no tiene implementación. Una clase con uno o más métodos *abstract* debe ser declarada a su vez como *abstract*.

No se pueden instanciar objetos de una clase *abstract*, pero sí podemos heredar de ella. Las subclases deberán implementar los métodos *abstract* o bien ser declaradas a su vez como *abstract*. Para implementar un método `abstract` no hay más que definirlo normalmente (quitando la palabra reservada *abstract* de la cabecera):

```
class Circulo extends ObjetoGrafico
{
      void dibujar()
{
            ...
      }
}


class Rectangulo extends ObjetoGrafico
{
      void dibujar()
{
            ...
      }
}
```

La ventaja de haber declarado *draw()* en *ObjetoGrafico*, aun sin haberlo implementado, estriba en lo siguiente: podemos declarar variables de tipo `ObjetoGrafico` (aunque los objetos asignados van a tener que ser de alguna subclase), e invocar con dicha referencia el método *draw()*. Podemos tener por ejemplo un *array* de `ObjetoGrafico` y con un sencillo bucle dibujar todos los objetos en él contenidos. El *enlace dinámico* nos garantiza que para cada objeto se ejecutará el método `draw()` adecuado.

6.4.9 Interfaces

Java no dispone de herencia múltiple. Eso significa que no podemos heredar directamente de más de una clase. A veces sería conveniente hacerlo, bien sea por motivos conceptuales o prácticos. Un ejemplo del primer caso es el siguiente: tenemos una clase *Estudiante* (con atributos como las asignaturas que está haciendo) y una clase *Profesor* (con

atributos como el sueldo que cobra de la universidad). Nos gustaría poder definir una clase *Becario* que tuviera los atributos y métodos de ambos.

Desearíamos disponer de herencia múltiple por razones prácticas en el siguiente caso. Imaginemos que queremos que los directivos de nuestra empresa puedan ser ordenados por sueldo. Alguien ha escrito una rutina de ordenación de *arrays*, pero dicha rutina necesita de un método `compareTo()` para comparar los elementos del *array*:

```
abstract class Sortable {
      public abstract int compareTo(Sortable b);
}

class ArrayAlg
{
      public static void sort(Sortable[] a) {
            ...
      }
}
```

La rutina de ordenación sólo funciona para *arrays* de tipo *Sortable* (o cualquiera de sus descendientes), puesto que es la forma de asegurarse que todo objeto en el *array* dispone del método *compareTo()*. Para usar esta rutina de ordenación para ordenar directivos, la clase *Directivo* tendría que heredar de *Sortable*, pero ya hereda de *Empleado*.

En estos casos Java ofrece una facilidad que nos permite lograr casi los mismos resultados que si tuviéramos herencia múltiple: las *interfaces*.

Una interfaz es como un sucedáneo de clase que "promete" la disponibilidad de unos ciertos métodos sin implementar ninguno de ellos. Se declara de forma muy similar a una clase:

```
public interface NombreInterface
{
      ...
}
```

Dentro podemos declarar cabeceras de métodos y constantes. De alguna forma es como una clase totalmente abstracta. El punto clave es que una clase puede heredar de una única clase, pero puede "implementar" tantos *interfaces* como le haga falta. Y cuando una clase declara que "implementa" un cierto interfaz, se interpreta a todos los efectos que dicha clase también posee el tipo del interfaz. Así si *Sortable* fuera un interfaz en lugar de una clase, podríamos heredar *Directivo* de *Empleado* como hasta ahora, y declarando que *Manager* implementa *Sortable* podríamos pasar un *array* de *Directivo* a la rutina `sort()`:

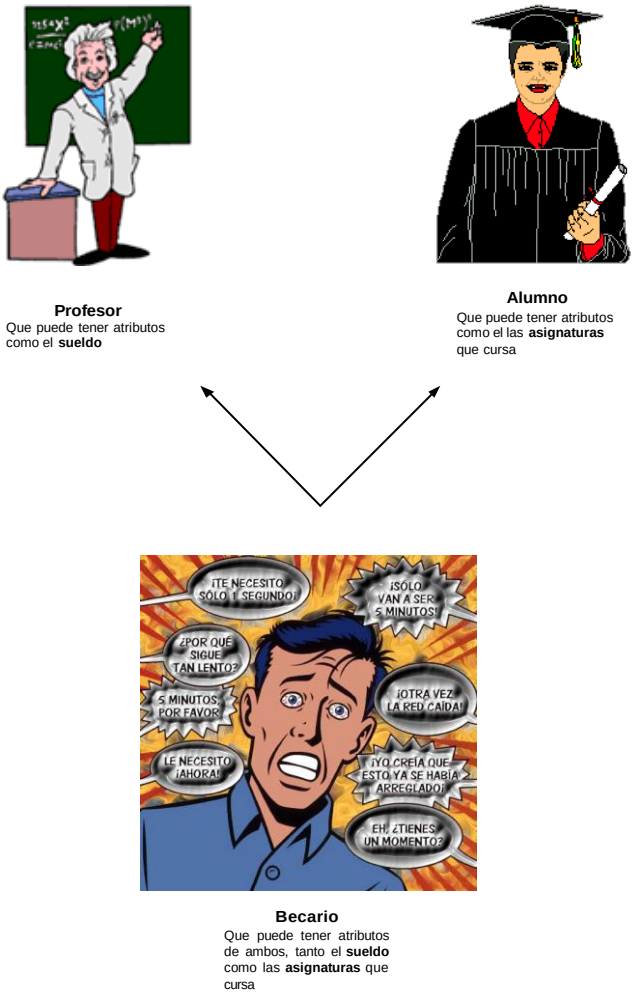


Fig. 12
Herencia múltiple

```
public interface Sortable
{
      public int compareTo(Sortable b);
}

public class Directivo extends Empleado implements Sortable
{
      ...

      public int compareTo(Sortable b)
{
            ...
      }
}
```

Para indicar los *interfaces* implementados usamos *implements*:

```
class NombreClase extends NombreSuperClase implements Interface1,
Interface2 ...
{
      ...
}
```

Podemos declarar variables de tipo *interface*, pero no instanciar objetos de tipo *interface* (idéntico que con las clases abstractas). También es posible que un *interface* herede de otro mediante extends.

Como conclusión podemos decir que en Java sólo podemos heredar la implementación de un único tipo, pero podemos heredar la interfaz (es decir, una lista de métodos con su cabecera) de varios tipos a la vez.

6.4.10 Simulando la genericidad mediante herencia

Una facilidad que ofrecen algunos lenguajes orientados a objetos es la de poder definir tipos parametrizados (con atributos y otras referencias cuyo tipo es arbitrario). Esto es muy útil para clases de tipo colección de objetos (contenedores). Por ejemplo, nos gustaría poder definir una clase *Lista* con un parámetro que fuera el tipo de los objetos que contiene. Podríamos instanciar listas de *Empleado* o listas de *String*, o de otros tipos según nos conviniera, sin modificar para nada el código de la clase *Lista*. Este mecanismo se conoce como genericidad y por desgracia Java no lo soporta.

No obstante, es posible lograr resultados parecidos de la siguiente manera: si hacemos una lista de *Object*, la lista podrá contener cualquier objeto, sea cual sea su tipo (todas las clases heredan directa o indirectamente de *Object*). Esto se utiliza en todas las colecciones y en cualquier otro tipo que queramos hacer genérico.

Lógicamente el compilador no nos obligará a que los objetos insertados en la lista sean de ningún tipo en especial (no podremos declarar una lista de *Empleado* donde sólo sea posible insertar objetos *Empleado*). Por otro lado todo lo que saquemos de la lista será de tipo *Object* y habrá que hacer el *Object* al tipo adecuado para poder trabajar con el objeto. Será responsabilidad del programador insertar sólo objetos del tipo correcto.

Los únicos tipos que no podremos insertar en nuestra lista son los tipos primitivos, puesto que al no definirse a partir de clases, no heredan de *Object*. ¿Cómo hacer una lista de `int`? Para estos casos existen unas clases especiales llamadas *wrappers* (envoltorios) para encapsular un valor de un tipo primitivo en un objeto. Hay un *wrapper* para cada tipo primitivo: `Integer`, `Long`, `Float`, etc. Ejemplo:

```
new Integer(254)
```

crea un objeto que puede ser usado en cualquier lugar donde se espere un *Object*. Después el valor contenido se puede recuperar invocando el siguiente método:

```
int intValue()
```

Estas clases también proporcionan métodos útiles para trabajar con tipos primitivos, como por ejemplo obtener un número a partir de un `String` que contiene dígitos.

En *java.util* hay un número importante de colecciones agrupadas en varias familias:

Collection: un grupo de objetos sin un orden específico.
Set: un grupo de objetos sin elementos repetidos.

SortedSet: un grupo de objetos sin elementos repetidos y recorrible en un cierto orden.
List: un grupo de objetos ordenado. Se permiten duplicaciones.
Map: un grupo de objetos identificados por una clave.
SortedMap: un grupo de objetos identificados por una clave recorrible en un cierto orden.

Cada familia viene a representar un TAD diferente, definido por las operaciones básicas aplicables al mismo. La lista mostrada es de *interfaces*.

El API (en la documentación de cada interfaz se muestran todas las clases conocidas que las implementan) contiene también clases como *HashSet*, *ArraySet*, *ArrayList*, *LinkedList*, *Vector*, etc., que implementan estas *interfaces*, dando implementaciones alternativas para cada TAD. El API proporciona métodos con algunos algoritmos como la ordenación, la búsqueda binaria o la evaluación del máximo o el mínimo, y clases con una funcionalidad más específica como *Stack*.

La clase *Vector* crece dinámicamente y permite insertar elementos de tipos diferentes. Tiene acceso directo por posición (empezando por el 0) pero a la vez permite insertar en cualquier posición desplazando todos los elementos necesarios hacia la derecha.

Entraremos un poco más en detalle con la funcionalidad de esta clase.

```
janva.lang.Object
└── java.util.AbstractCollection
    └── java.util.AbstractList
        └── java.util.Vector
```

```
public class Vector extends AbstractList implements List,
      Cloneable, Serializable
{
...
}
```

Cada vector tiene las variables de *capacity* y *capacityIncrement*. A medida que se añaden elementos al vector, la memoria reservada para éste aumenta en términos que marca la variable *capacityIncrement*.

La capacidad de un vector es, como mínimo, tan grande como la longitud del vector (normalmente mayor).

Los constructores de la clase Vector son:

public Vector(). Construye un vector vacío.
public Vector(int initialCapacity). Construye un vector vacío de una determinada capacidad.
public Vector(int initialCapacity, int capacityIncrement). Construye un vector vacío, con una capacidad determinada y un valor de capacityIncrement determinada.

A continuación se explican algunos de los métodos de la clase Vector. Para conocer la descripción de todos los métodos de esta clase, se puede mirar el API de las librerías del J2SE:

public final int size(). Retorna el número de elementos en el vector (no es lo mismo que la capacidad del vector).
public final boolean contains(Object elem). Retorna true si el objeto especificado es un valor de la colección.
public final int indexOf(Object elem). Busca el objeto especificado desde la primera posición y retorna un índice (o -1 si el elemento no está). Utiliza el método equals(), de modo que si el objeto no tiene sobrescrito el método equals() de la clase Object, sólo comparará los punteros, no los contenidos.
public final synchronized Object get(int index). Retorna el elemento que corresponde al índice que se le pasa. Si el índice es incorrecto, lanza la excepción: *ArrayIndexOutOfBoundsException* (se hablará de excepciones en el siguiente capítulo).
public final synchronized void set(int index, Object obj). Reemplaza el elemento que corresponde al índice, por el objeto que se le pasa. Si el índice no es correcto, lanza la excepción: *ArrayIndexOutOfBoundsException*.
public final synchronized void remove(int index). Borra el elemento especificado por el índice. Si el índice no es válido, lanza la excepción: *ArrayIndexOutOfBoundsException*.
public final synchronized void add(Object obj). Añade el objeto que se le pasa, en la última posición.
public final synchronized void add(int index, Object obj). Inserta el objeto que se le pasa en la posición que indica el índice, pasando hacia arriba todos los elementos

con un índice igual o superior. Si el índice es inválido, lanza la excepción: *ArrayIndexOutOfBoundsException.*

> **Ejercicio 6.6**
>
> Basándoos en la clase *PruebaDirectivo* del ejercicio 8, sustituid todos los *arrays* que utilizastéis por objetos de tipo *Vector*.

6.4.11 Agrupación de clases

Existe una forma de agrupar las clases para organizar el código fuente de nuestros proyectos y bibliotecas de componentes reutilizables: es lo que se denomina *package*. Las clases de librería del J2SDK se distribuyen en numerosos *package*. Por ejemplo, la clase *GregorianCalendar* se encuentra en el *package* java.util, y la clase *Math* está en java.lang. Los *package* se organizan jerárquicamente, de forma análoga a los directorios. El *package* de una clase se declara al inicio del fichero fuente, por ejemplo:

```
package miempresa.miproyecto.gui;
```

Si no se declara *package* se supone que la clase se encuentra en el *package* raíz. Si usamos clases que se encuentran en otro *pakcage*, cuando usemos el nombre de la clase deberá precederlo su camino en la jerarquía de *packages*:

```
java.util.GregorianCalendar  fechaActual =
                    new java.util.GregorianCalendar();
```

a menos que después de la declaración del *package* indiquemos qué clases de otros *packages* vamos a usar:

```
import java.util.GregorianCalendar;
```

También es posible importar todas las clases de un *package* (no incluye posibles *subpackages* de forma recursiva):

```
import java.util.*;
```

No es necesario importar java.lang; es el único *package* que se encuentra siempre disponible.

No puede haber dos clases con el mismo nombre dentro de un mismo *package*.

Los miembros para los que no declaramos visibilidad, no son en realidad ni *public*, ni *protected*, ni *private*, sino que tienen una visibilidad restringida a las clases de su *package*. De hecho, para que una clase sea utilizable fuera de su *package* deberá declararse también *public* con

```
public class NombreClase ...
```

En caso contrario la clase no podrá importarse desde fuera.

Los miembros *protected* también son visibles a todas las clases del *package*, aunque no sean descendientes de la clase donde se han definido.

Ejercicio 6.7

Vamos a hacer una aplicación que modelice el funcionamiento de un banco.

1. Crear una clase *CuentaBancaria* con los atributos *numeroCuenta* (que se pasará en el constructor) y *saldo*. Implementad los métodos para consultar estos atributo y otros para ingresar y retirar dinero y para hacer un traspaso de una cuenta a otra. Para los tres últimos métodos puede utilizarse un método privado *añadir(double cantidad)* que añada una cantidad (positiva o negativa) al saldo. También habrá un atributo común a todas las instancias *interesAnualBasico*, que en principio puede ser una constante. La clase tiene que ser abstracta y debe tener un método *calcularIntereses()* que se dejará sin implementar. De esta clase heredarán dos: *CuentaCorriente* y *CuentaAhorro*. La diferencia entre ambas será la manera diferente de calcular los intereses. A la primera se le incrementará el saldo teniendo en cuenta el interés anual básico. La segunda tendrá una constante de clase s*aldoMinimo*. Si no se llega a este saldo el interés será la mitad del interés básico, si se supera el doble. Haced una clase *PruebaCuentas* para comprobar las tres clases.

2. Haced una clase *ClienteBanca* con un nombre, dni y *Vector* de cuentas bancarias de los cuales es titular. Implementad métodos *añadirCuenta* y *obtenerCuenta*. Añadid también un atributo *titular* en *CuentaBancaria*, que apuntará al cliente titular de la cuenta.

3. Implementad la clase *Banco* de manera que tenga una tabla de cuentas indexada por *numeroCuenta* y una tabla de clientes indexada por *dni*. Se puede utilizar la clase *java.util.HashMap*. Implementad el método *abrirCuentaCorriente()* y *abrirCuentaAhorro()* que creen una cuenta y la asocien a un cliente. Para que no haya cuentas con el mismo número *Banco* deberá mantener un atributo *numeroCuentas* que se vaya incrementando.

4. Implementad el método *listarClientes* que devuelva simplemente los clientes en forma de *String*. Usad *java.util.Iterator* Ídem para el método *listarCuentas* en la clase *ClienteBanca* usando *java.util.Enumeration*.

5. Implementad, en este orden, los métodos *ingresar ()*, c*onultarSaldo(), retirar ()* y *traspasar()* identificando a la cuenta por su número.

6. Implementad *modificarInteres()*. Ahora el atributo *interesAnual* de la clase *CuentaBancaria* no puede ser una

constante, pero tiene que seguir siendo un valor compartido para todas las instancias de la clase (y las subclases).

7. Haced que el *Banco* no sea el encargado de calcular el próximo número de cuenta, sino que sea la propia clase *CuentaBancaria*.

8. Implementad *cerrarCuenta()* en *Banco*. Deben desaparecer la referencia que tiene el banco y la que tiene el titular. Sería aconsejable añadir un método *quitarCuenta()* en (si no se ha hecho ya) en *ClienteBanca*.

9. Haced que el listado de clientes se haga en orden alfabético por nombre, haciendo que cliente bancario implemente la interfaz *Comparable* y guardando los clientes en un *TreeMap*.

7 EXCEPCIONES

Las excepciones son un mecanismo que presentan los lenguajes más modernos para ejecutar código aparte del control de flujo habitual cuando se producen ciertos errores o condiciones anormales de ejecución.

Constituyen una alternativa menos tediosa y más segura que control de errores clásico para escribir aplicaciones robustas.

7.1 Introducción

Vamos a ver cómo se definen exactamente las excepciones en Java y cómo se utilizan.

7.1.1 ¿Que es una excepción?

En el lenguaje Java, la clase *Exception* define condiciones de error que los programas pueden encontrar. En vez de dejar que el programa termine, se puede escribir código para gestionar estas condiciones de error y continuar con la ejecución del programa.

Cualquier situación anormal que rompe la ejecución normal del programa se denomina error o excepción. Las excepciones se producen por ejemplo cuando:

El fichero que se quiere abrir no existe.
La conexión de red se ha perdido.
Los operandos que se manejan se salen de rango.
El fichero de clase que se quiere cargar no existe.

7.1.2 ¿Qué es un error?

En el lenguaje Java, la clase `Error` define aquellas condiciones que se consideran como errores serios de los que es preferible no recuperar la ejecución. La mayoría de veces es aconsejable dejar que el programa termine.

7.1.3 Primer ejemplo

El lenguaje Java implementa excepciones al estilo de C++ para ayudar a escribir código robusto. Cuando se genera un error en un programa, el método que encuentra dicho error "lanza" una excepción de vuelta al método que lo invocó. El resto de código después de la instrucción que produjo la excepción no se ejecutará. El segundo método puede capturar esa excepción y si es posible corrige la situación de error y continúa con la ejecución. Este mecanismo proporciona la posibilidad al programador de escribir un gestor que trate con la excepción.

Consultando la documentación del API se pueden ver qué excepciones lanzan los diferentes métodos.

Considerar esta nueva versión de *HolaMundo* que imprime mensajes cíclicamente hasta leer más allá de los índices:

```
public class HolaMundoExcepcion
{
      public static void main(String[] args)
      {
            String[] mensaje = new String[2];

            mensaje[0]="Hola ";
            mensaje[1]="mundo!";

            // este bucle accedera a un indice fuera de rango
            // y producira un error
            for(int i=0;i<3;i++)
                  System.out.println(mensaje[i]);
      }
}
```

Normalmente un programa termina con un mensaje de error cuando se lanza una excepción durante su ejecución, tal y como el ejemplo anterior ha hecho al ejecutar el bucle por tercera vez:

```
Hola
mundo!
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
        at HolaMundoExcepcion.main(HolaMundoExcepcion.java:13)
```

El manejo de excepciones permite al programa capturar las excepciones, gestionarlas y entonces continuar con su ejecución. El código se estructura de manera que los casos de error no interfieran con la ejecución normal del programa. Estos casos de error se gestionan en bloques de código separados. De este modo se obtiene un código más manejable y legible.

7.2 Manejo de excepciones

El lenguaje Java proporciona un mecanismo para averiguar qué excepción se ha lanzado y cómo recuperarse de ella.

7.2.1 Sentencias try y catch

Para capturar una excepción en particular hay que colocar el código que puede lanzar dicha excepción dentro de un bloque *try*, entonces se crea una lista de bloques *catch* adyacentes, uno para cada posible excepción que se quiera capturar. El código de un bloque *catch* se ejecuta cuando la excepción generada coincide en tipo con la del bloque *catch*.

```
try
{
// codigo que puede lanzar una execpcion en particular
}
catch (MiExcepcion e)
{
// codigo a ejecutar si se lanza una excepcion
// MiExcepcion
}
catch (Exception e)
{
      // codigo a ejecutar si se lanza una excepcion generica
      // Exception
}
```

7.2.2 Sentencia finally

La sentencia *finally* define un bloque de código que se ejecuta siempre, independientemente de si una excepción fue capturada o no.

```
try
{
      Sytem.out.println("Intentamos  aprobar el examen de Java");
      examen.aprobar(yo);
}
catch (SuspenderException e)
{
      System.out.println("Vaya,  algo no ha ido bien..."+e);
}
finally
{
      System.out.println("Haya  aprobado o no, los profes son
            geniales");
}
```

En el ejemplo anterior intentamos aprobar un examen, en caso de no conseguirlo se muestra un mensaje indicándolo (fijaos que se puede imprimir la excepción concatenándola ya que como clase que es dispone de su método *toString()*), pero en cualquier caso alabamos a los profesores porque son excepcionales.

El código entre llaves de la sentencia *try* se llama código protegido. La única situación en la que la sentencia *finally* puede no ejecutarse es si se llama al método *System.exit()* , que termina con la ejecución del programa, dentro del código protegido. Esto implica que el flujo normal de ejecución del código se puede desviar del orden secuencial. Por ejemplo, si existiese una sentencia *return* dentro del código protegido, el bloque *finally* se ejecutaría antes que la sentencia *return*.

En caso de que no hubiese ninguna alteración del flujo de ejecución, tras el código del bloque *finally* se ejecutaría lo que hubiese después.

7.2.3 El mecanismo de pila de llamadas

Si una sentencia de un método lanza una excepción que no se gestiona en el correspondiente bloque try/catch, la excepción se propaga al método inmediatamente superior. Si en éste no es tratada, se continua propagando a los métodos superiores. Este proceso se repite hasta que la excepción es gestionada. Si una excepción no se gestiona y llega hasta el método *main(...)* y ni siquiera éste la gestiona entonces se produce un final anormal de la ejecución del programa (como hemos visto ya en un ejemplo).

Si se considera un ejemplo donde el método *main(...)* llama a otro método (por ejemplo *primero()*) y éste a su vez llama a otro método (por ejemplo *segundo()*). Si ocurre una excepción en *segundo()* se comprobará si hay un *catch* para dicha excepción; sino, se buscará en el siguiente método de la pila de llamadas (*first()*), y después el siguiente (*main()*). Si la excepción no es gestionada por este último método de la pila, se produce un error de ejecución y el programa termina.

En caso de que en algún punto de la pila de llamadas se capture la excepción, se ejecutará el bloque *catch* correspondiente que será el encargado de solucionar el problema o de comunicarlo. La excepción dejará de propagarse hacia arriba.

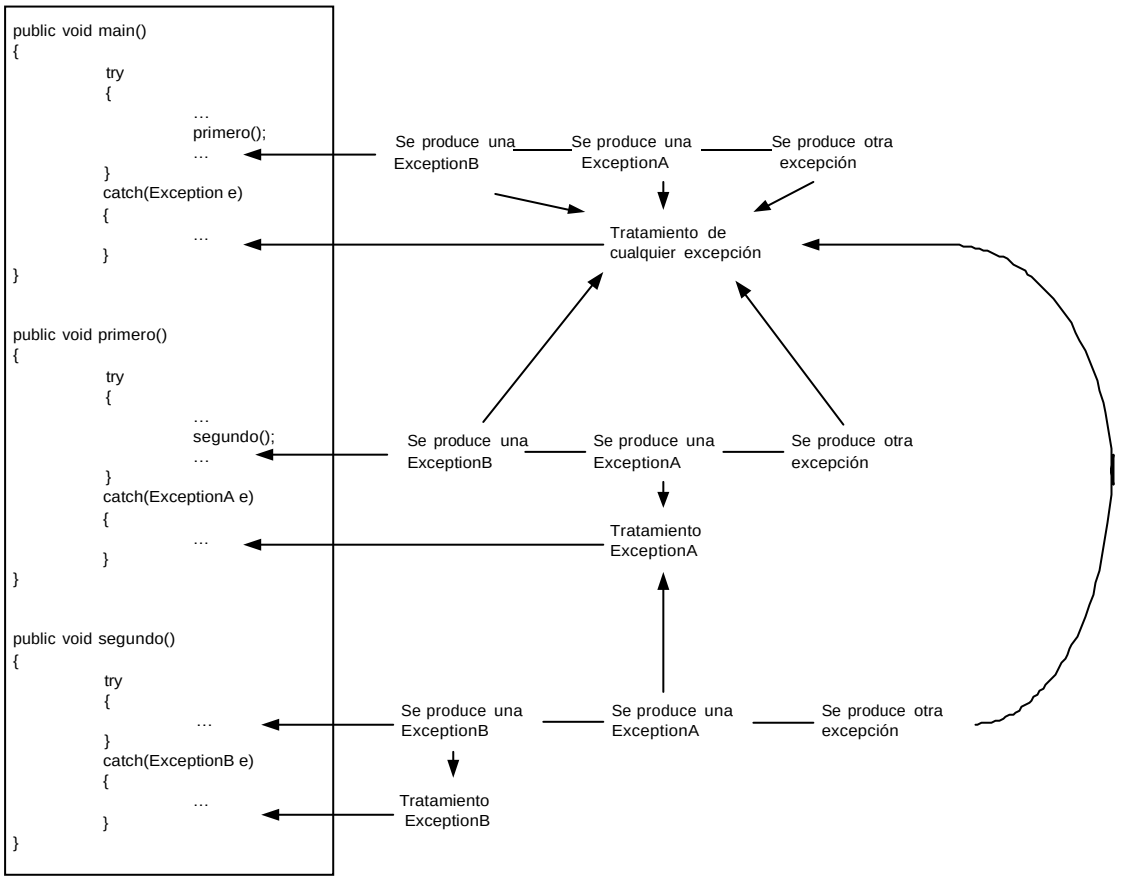


Fig. 13
Tratamiento de excepciones

7.2.4 Categorías de excepciones

En el lenguaje Java existen tres categorías genéricas de excepciones. La clase *java.lang.Throwable* que actúa como clase padre de todos aquellos objetos que pueden ser lanzados y capturados utilizando los mecanismos de gestión de excepciones. Se han definido una serie de métodos en la clase *Throwable* para recuperar los mensajes de error asociados con la excepción, y para imprimir una traza de la pila para saber donde ha ocurrido la excepción. Existen dos subclases de *java.lang.Throwable* **,** *Error* y *Exception*, tal y como se muestra en el diagrama:

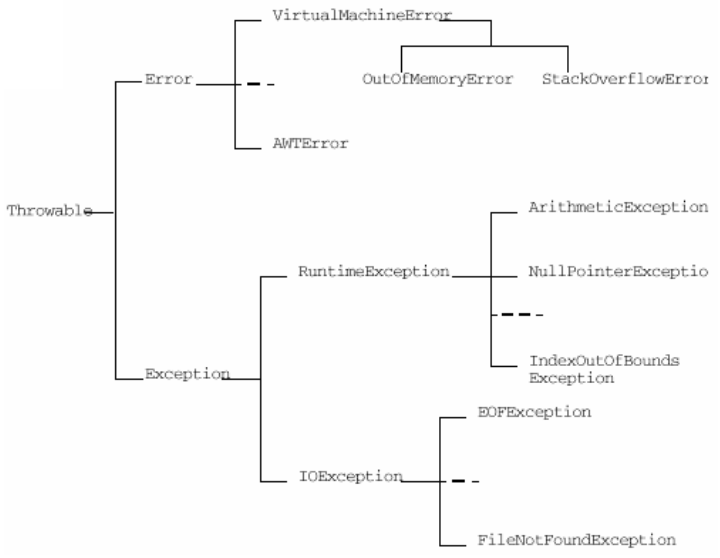
La clase Throwable no debería usarse directamente, sino una de las que se derivan de ella. El propósito de cada tipo de excepción es el siguiente:

Las clases que descienden de *Error* indican un problema bastante serio del cual la recuperación es prácticamente imposible Un ejemplo es la ejecución sin memoria No se espera que un programa pueda gestionar este tipo de condiciones si no es saliendo de la aplicación lo más elegantemente posible.

RuntimeException y descendientes sirven para indicar un error de diseño o implementación del programa. Es decir, indica condiciones que no deberían pasar nunca si se programa cuidadosamente. La excepción *ArrayIndexOutOfBoundsException* no debería lanzarse nunca si los índices de acceso a un vector no se salen de los límites. Puesto que un buen diseño e implementación de un programa no lanza nunca este tipo de excepciones, no se acostumbra a gestionarlas. Esto se traduce en un mensaje de error en tiempo de ejecución, y asegura que el programador tomará las medidas necesarias, en vez de ocultar el error. Son útiles pues para refinar.

Otras excepciones (las que heredan de *Exception* pero no de *RuntimeException*) indican una dificultad en tiempo de ejecución debido a errores en el entorno de ejecución o de usuario. Un ejemplo claro de esto puede ser un fichero no encontrado. Debido a que esto puede ocurrir debido a un error de usuario (introducción de nombre incorrecto) se recomienda a los programadores que capturen y gestionen este tipo de excepciones.

Fig. 14
Jerarquía de excepciones



7.2.5 Excepciones más frecuentes

El lenguaje Java proporciona una serie de excepciones predefinidas. A continuación se citan las más comunes:

ArithmeticException. Es el resultado de dividir por cero números enteros:

```
int i = 12 / 0;
```

NullPointerException. Intento de acceder a los atributos o métodos de un objeto cuando todavía no está instanciado:

```
Date d = null;
System.out.println(d.toString());
```

NegativeArraySizeException. Intento de creación de un vector con un número negativo de elementos.
ArrayIndexOutOfBoundsException. Intento de acceso a un elemento de un *array* fuera de rango.
SecurityException. Error típico en los navegadores; La clase *SecurityManager* lanza esta excepción cuando un *applet* intenta hacer lo siguiente (salvo que se le permita explícitamente):

- Acceder a un archivo local.
- Abrir un *socket* con un *host* diferente del que sirvió el *applet*.
- Ejecutar orto programa en el entorno de ejecución.

7.2.6 La regla de "declarar o capturar"

A continuación de la palabra reservada *throws* aparece una lista de todas las excepciones que se pueden dar dentro del método y no serán gestionadas por él. Aunque en el ejemplo del final del capítulo sólo hay una, se pueden indicar varias excepciones separándolas por comas.

El escoger si gestionar o declarar la excepción depende de quién se considere como mejor candidato para gestionar la excepción. Es obligatorio declarar las excepciones no capturadas excepto para aquellas que descienden de Error o de *RuntimeException*.

Debido a que las clases de excepciones se organizan según una jerarquía, es posible capturar grupos de excepciones y gestionarlas a la vez con el mismo código de captura. Por ejemplo, aunque existen diferentes tipos de excepciones de *IOException* (*EOFException*, *FileNotFoundException* y demás), con capturar *IOException* estamos capturando cualquiera de las clases que descienden *IOException.*

Ejercicio 7.1

Haced una clase *Calculadora* que realice operaciones sencillas como suma, resta, multiplicación división de enteros. Haced que se lancen las excepciones oportunas en caso de operaciones conflictivas. Recordad lo que es un método estático y usadlos siempre que podáis.

Implementad ahora una clase *PruebaCalculadora* que se encargue de tratar las posibles excepciones que se puedan producir sin abortar la ejecución del programa. Ayudaos de la clase *Console* para realizar la entrada/salida de *PruebaCalculador*.

Ejercicio 7.2

Reprogramad ahora *Calculad ora* para que sea esta quien trate las excepciones.

7.3 Creación de excepciones de usuario

Hablaremos en este apartado de cómo crear y utilizar clases que representan nuevas condiciones de error definidas por nosotros mismos.

7.3.1 Introducción

Las excepciones definidas por usuario se crean heredando de la clase *Exception*. Estas clases pueden contener cualquier cosa al igual que una clase estándar. A continuación se muestra un ejemplo de excepción definida por el usuario, con su constructor, algunas variables y métodos:

```
public class ServidorTimedOutException  extends Exception
{
      private String motivo;
      private int puerto;

      public ServidorTimedOutException  (String motivo,int puerto)
      {
            this.motivo = motivo;
            this.puerto = puerto;
      }

      public String getMotivo()
      {
            return motivo;
      }

      public int getPuerto()
      {
            return puerto;
```

```
      }
}
```

Para lanzar una excepción creada por el usuario se utiliza la sentencia *trow* como se muestra en el ejemplo siguiente:

```
throw new ServidorTimedOutException("Imposible conectar",80);
```

7.3.1.1 Ejemplo

Considerar un programa cliente-servidor. En el código del cliente se intenta una conexión con el servidor y se espera una respuesta en un tiempo máximo de 5 segundos. Si el servidor no responde, el código podría lanzar una excepción (como por ejemplo `ServidorTimedOutException`) del siguiente modo:

```
public void conectame(String nombreServidor) throws
                                    ServidorTimedOutException
{
int exito;
      int puertoParaConectar = 80;

      exito = open(nombreServidor, puertoParaConectar);

      if (exito == -1)
{
        throw new ServidorTimedOutException("Imposible conectar", 80);
      }
}
```

Para capturar la excepción utilizar la sentencia *try*:

```
public void buscarServidor()
{
      ...
      try
{
            conectame(defaultServer);
      }
catch(ServidorTimedOutException e)
{
      System.out.println("Imposible conectar, probando
                                                      alternativa");
            try
            {
                  conectame(alternateServer);
            }
      catch (ServidorTimedOutException e1)
            {
                  System.out.println("No hay ningun servidor disponible
                                                      acutalmente");
            }
      }
...
}
```

Los bloques *try* y *catch* se pueden anidar tal y como se muestra en el ejemplo anterior.

También es posible gestionar parcialmente una excepción antes de volverla a lanzar. Por ejemplo:

```
try
{
...
}
catch (ServerTimedOutException e)
{
      System.out.println("Error caught ");
      throw e;
}
```

Ejercicio 7.3

Partiendo del banco del ejercicio 10, hacer que la Clase *CuentaBancaria* lance excepciones del tipo *SaldoInsuficienteException* que tenga los atributos y métodos necesarios para conocer el saldo disponible, la cantidad pedida y la diferencia entre ambos; si la cuenta no está actualmente (antes de intentar retirar el dinero) en numeros rojos el banco puede conceder un crédito y se lo indica al cliente mediante un *String,* en caso contrario se le indica la imposibilidad de retirar ese dinero. La clase *Banco* debe lanzar excepciones del tipo *CuentaInexistenteException*. Decidid para cada caso qué clase debe tratar la excepción.

Ejercicio 7.4

Cread la clase *Conjunto* de int implementándola con un **array** (obviamente la clase conjunto ya está implementada en la api de Java pero si la usáis el ejercicio pierde la gracia) y dotadla de un único método público

public void insertarInt(int i)

Los conjuntos no admiten elementos repetidos, por lo que cuando intentemos insertar uno ya preexistente lanzaremos una excepción del tipo *ElementoRepetidoException*, clase que debe contener métodos para consultar cuál ha sido el elemento que ha producido el lanzamiento de la misma.

NOTA: los conjuntos en realidad no funcionan así: cuando se inserta un elemento preexistente simplemente no pasa nada, es decir, no se inserta un elemento repetido pero tampoco se produce un error. Usamos este "conjunto modificado" simplemente para ilustrar los conceptos que estamos tratando en este capítulo

8 PROGRAMACIÓN GRÁFICA. EL ENTORNO SWING

Antes de Java 2 las aplicaciones gráficas se programaban a través del GUI (Graphical User Interface) conocido como *AWT* (Abstract Window Toolkit). El principal problema de esta GUI es que no estaban programadas sobre objetos Java directamente y por tanto su apariencia (entre otras cosas) dependía de la arquitectura sobre el que se estaba ejecutando.

Swing es la solución que se dio a este problema en Java 2 ya que está totalmente programado en Java puro, así pues nuestras aplicaciones gráficas se visualizaran de idéntica manera en un Solaris que en un Windows o en un Linux; de todos modos, si deseamos forzar que la apariencia sea la de un entorno concreto o la del entorno sobre el que se ejecuta, también puede hacerse gracias a unas funcionalidades que incorpora y que recibe el nombre de *Look & Feel.*

Swing es uno de los componentes más importantes de las JFC (Java Foundation Classes) que se desarrollaron gracias a un esfuerzo de cooperación entre importantes firmas del mundo informático, tales como Sun Mycrosystems, Netscape o IBM. A pesar de esto ninguna de las versiones actuales de navegadores soporta directamente la ejecución de *applets* desarrollados con versiones Java 2 y es necesario descargar un plug-in al efecto.

Swing, a pesar de estar totalmente programado en Java, se apoya en *AWT* que es el que proporciona el interfaz entre el sistema de ventanas nativo de la arquitectura y la GUI de Java.

8.1 Primer ejemplo

Para todo aquel que haya programado aplicaciones gráficas todo este capítulo le resultará bastante familiar, aunque no usaremos ningún entorno de desarrollo gráfico que nos ayude. Esto, sin duda, dificulta sensiblemente la labor y la hace más críptica.

Para empezar a familiarizamos con el entorno gráfico crearemos una primera aplicación, que abrirá una ventana en la pantalla. Las ventanas principales (que no dependen de otra ventana) se llaman *frames* o marcos. Utilizaremos la clase *JFrame*. Si os miráis la documentación de la API a fondo veréis que existe una curiosa duplicidad de las clases que representan objetos gráficos, así pues tenemos un *Frame* y un *JFrame*, un *Button* y un *JButton*, etcétera. Esto se debe a que los objetos sin *J* corresponden a las versiones AWT (que no usaremos por obsoletas y dependientes de la plataforma) y los objetos con *J* son los correspondientes *Swing*.

```
01    import javax.swing.*;
02
03    public class HolaMundoConVentanas
04    {
05          public static void main(String args[])
06          {
07                JFrame miFrame = new JFrame("Hola mundo ventanas!");
08                miFrame.setSize(300,300);
09                miFrame.setVisible(true);
10          }
11    }
```

Cuando declaramos el marco y lo dimensionamos no es suficiente para que aparezca hay que forzarlo como se muestra en la línea 9.

Un detalle importante es que no hemos programado el tratamiento de eventos porque lo veremos más adelante, por eso al intentar cerrar la ventana aunque parezca que esta desaparece (que no es cierto, sólo se hace invisible), el programa sigue activo. Hasta que sepamos cómo tratar los eventos para cerrar la ventana no nos quedará más remedio que matar la ejecución de nuestro programa Java (CTRL+C en la mayoría de sistemas operativos).

8.2 Modo gráfico

Aquí veremos cómo se escribe y se dibuja en modo gráfico.

8.2.1 Texto y fuentes

Ahora que podemos crear una ventana, vamos a empezar por escribir un texto en ella. Crearemos una clase *MiFrame* que herede de *JFrame*.

Para ello, además de los métodos anteriores, también habremos de sobrescribir el método *paint(Graphics g)* que se hereda de *java.awt.Component* (se hablará de esto más adelante) y allí especificaremos lo que queremos dibujar (en este caso escribir).

El método *paint(Graphics g)* es el encargado de pintar los componentes en la pantalla, de tal manera que será el método que se invoque cada vez que nuestra aplicación necesite ser redibujada. Por ejemplo, cuando pongamos otra ventana encima y luego la quitemos, el trozo destruido de nuestra aplicación (el trozo ocultado por la otra ventana) necesitará ser redibujado. De esto se encarga siempre dicho método.

La particularidad de este método es que no lo llamaremos directamente mediante una invocación corriente, sino que lo haremos a través de una llamada al método *repaint()*, que será el encargado de hacer los cálculos pertinentes para reparar la zona de pantalla que ha sido destruida y hacer así que sólo se redibuje lo necesario; ganando así en velocidad de refresco y evitando efectos de parpadeo.

La manera de escribir texto (y dibujar) consiste en manipular el parámetro de tipo Graphics que recibe. Por ejemplo:

```
import javax.swing.*;
import java.awt.*;

public class MiFrame extends JFrame
{
      public MiFrame()
      {
            super("¡Hola mundo con ventanas!");
            setSize(300,300);
            setVisible(true);
      }

      public void paint(Graphics g)
      {
            g.drawString("Tenemos los mejores profes de Java",40,160);
      }

      public static void main(String args[])
      {
            MiFrame mf = new MiFrame();
      }
}
```

El método *drawString(...)* tiene tres parámetros, el `String`, y las coordenadas x e y. Estas coordenadas se consideran a partir de la esquina superior izquierda. Cada vez que se tiene que refrescar la pantalla, Java llama al método *update()* (de la clase *Component*), que borra la pantalla y luego éste llama a *paint()*, que es donde definimos usualmente lo que deseamos dibujar.

Figura 15
Primer programa con ventanas

Cuando escribimos con la llamada a *drawString(...)*, la fuente usada es la que esté activa. Podemos definir una fuente con el tipo de letra, el estilo y el tamaño. Luego asignaremos esta fuente al entorno gráfico (figura .

```
Font f = new Font("Times Roman", Font.BOLD + Font.ITALIC, 12);
g.setFont(f);
```

Los estilos permitidos, definidos como constantes estáticas de la clase *Font*, son:

`Font.PLAIN`,
`Font.BOLD`,
`Font.ITALIC` y
`Font.BOLD` **+** *`Font.ITALIC`*

Una vez hemos definido la fuente que utilizamos, podemos recurrir a la clase *FontMetrics* para averiguar el tamaño que ocuparán las letras en nuestra ventana (en pixels).

```
FontMetrics fm = g.getFontMetrics(f);
```

Los métodos que encontraremos en la clase FontMetrics:

int getAscent (): distancia *baseline* basta la altura de las mayúsculas.
int getDescent (): distancia *baseline* hasta profundidad de los caracteres que descienden.
int getLeading(): espacio mínimo entre líneas.
int qetHeighc(): distancia entre dos líneas (*descent* + *leading* + *ascent*).
int getMaxAscent(): *ascent* máximo.
int getMaxDescent(): d*escent* máximo.
int stringWidth(String st): anchura de un *string* dado.

8.2.2 Líneas

Las líneas en Java se dibujan con el método *drawLine(...)*. En el ejemplo anterior sólo habría que añadir una instrucción al método *paint()*.

```
g.drawLine(5, 10,30,50);
```

Los parámetros son cuatro valores enteros (int) que corresponden respectivamente a las coordenadas *x* e *y* iniciales y finales.

8.2.3 Rectángulos

También existe un método específico para los rectángulos (hay que recordar que el cuadrado es un caso particular del rectángulo).

```
g.drawRect (x,y, anchura,altura);
```

Las variables x e y determinan la esquina superior izquierda y las otras dos la anchura y altura del rectángulo. Los cuatro valores son de tipo int.

Hay otros métodos que dibujan un rectángulo con las esquinas redondeadas, rectángulo en tres dimensiones y para que el rectángulo aparezca relleno.

```
void drawRoundRect(int x, int y, int width, int height, int arcwidtri,
     inc arcHeight)
void draw3DRect(int x, int y, inc width, int heiqht, boolean raised)
void fillRect(int x, int y, int width, int heiqht)
void fillRoundRect(int x, int y, int widch, int heiqht, int arcwidth,
     int arcHeight)
void fill3DRect(int x, int y, int width, int height, boolean raised)
```

8.2.4 Polígonos

Existe una función *drawPolygon(...)*, pero no es tan sencilla de usar como los rectángulos ya que el parámetro que recibe es un polígono o los *arrays* con las coordenadas.

```
void drawPolygon(Polygon P)
void drawPolygon(int[] xCoords, int [] yCoords, int n)
```

xCoords son las coordenadas x de los vértices, yCoords son las coordenadas y de los vértices y n es el número de vértices.

Se dibujarán líneas de tal manera que queden con los puntos (xCoords[i], yCords[i]) de punto incial y de punto final (xCoords[i+1], yCords[i+1])**.**

El polígono es un conjunto de segmentos que se cierra automáticamente. Con esto quiero decir que para dibujar un triángulo, por ejemplo, sólo hay que dar las coordenadas de los tres vértices y el sistema ya sabe que tiene que interpretar el último par de coordenadas y el primer par de coordenadas dados como el inicio y el fin respectivamente del segmento que cierra el polígono.

Para definir un polígono usamos el constructor y luego añadimos los puntos uno a uno.

```
Polygon p = new Polygon();
p.addPoint(x,y)
```

También encontramos una función *fillPolygon(...)* con el mismo comportamiento que las que hemos visto para los rectángulos.

8.2.5 Óvalos

Existe un método para dibujar elipses (y circunferencias), aunque Java las denomina óvalos.

```
void drawOval(int x, int y, int anchura, int altura)
```

Los cuatro parámetros del óvalo son las que delimitan el rectángulo que circunscribe la elipse. También existe el método *fillOval(...)*.

8.2.6 Arcos

La última forma geométrica que nos permite dibujar Java son los arcos.

```
void drawArc(int x, int y, int anchura, int altura,
                        int anguloInicial,int arcoAngulo)
```

Los cuatro primeros valores definen el rectángulo como en los óvalos, los dos últimos son el ángulo inicial y el ángulo del arco (relativo al ángulo inicial). El rectángulo es donde se encuadra el arco, el parámetro de *anguloInicial* indica posición desde qué posición se empieza a dibujar el arco y el *arcoAngulo* indica la porción de arco que hay que dibujar. Por ejemplo, para dibujar una

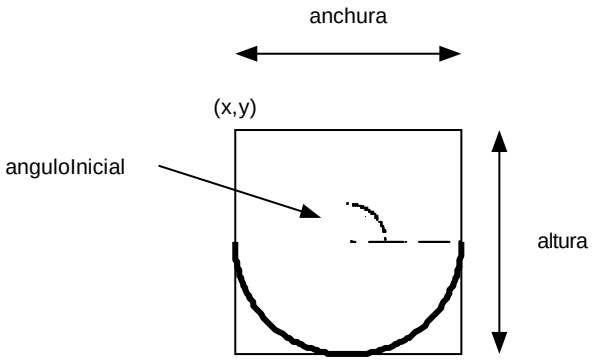


Figura 16
Uso de drawArc

semicircunferencia, el rectángulo deberá ser un cuadrado, a continuación fijar el ángulo inicial y luego decirle que dibuje 180 grados.

Cuando usamos el método *fillArc(...)*, la figura rellenada es el sector de la elipse delimitado por el arco y las líneas imaginarias que irían desde el centro de la elipse hasta los extremos del arco.

8.2.7 Colores

El texto que escribimos o las formas y figuras que dibujamos aparecen en el color que haya definido. Este color lo podemos modificar a nuestro gusto gracias al método setColor(...) de la clase Graphics, que recibe un parámetro de tipo Color.

Hay unas constantes en la clase Color que tienen definidos los valores *black , blue, cyan, darkGray, gray, green, lightcray, magenta, orange, pink, red, white* y *yellow.*

Si estos colores no nos bastan, podemos crear nuestro propio color a partir de sus valores RGB.

```
Color o = new Color(r,g,b)
```

Podemos cambiar el color de escritura de nuestra variable de gráficos y el color de fondo de nuestro contenedor (en nuestro ejemplo el Frame).

```
g.setColor(Color.red)
setBackground(Color.white);
```

8.2.8 Un ejemplo completo

```
import javax.swing.*;
import java.awt.*;

public class MiFrame extends JFrame
{
      public MiFrame()
      {
            super("¡Hola mundo con ventanas!");
            setSize(300,300);
            setVisible(true);
      }

      public void paint(Graphics g)
      {
            // Cambiamos el color del fondo del framse
            setBackground(Color.white);

            // Dibujamos texto
            Font f = new Font("Arial", Font.BOLD + Font.ITALIC, 14);
            g.setFont(f);
            g.drawString("Tenemos los mejores profes de Java",24,160);

            // Dibujamos líneas
            g.setColor(Color.red);
            g.drawLine(24,135,265,135);
            g.setColor(Color.blue);
            g.drawLine(24,175,265,175);
```

```
            // Dibujamos un rectángulo
            g.setColor(Color.black);
            g.drawRect(12,119,265,70);

            // Dibujamos un triángulo (polígono)
            int[] xCoords = new int[4];
            int[] yCoords = new int[4];

            xCoords[0]=150;
            yCoords[0]=35;
            xCoords[1]=100;
            yCoords[1]=100;
            xCoords[2]=200;
            yCoords[2]=100;

            g.setColor(Color.green);
            g.drawPolygon(xCoords,yCoords,3);


            // Dibujamos óvalos
            g.setColor(Color.orange);
            g.fillOval(107,200,85,85);
            g.setColor(Color.black);
            g.drawOval(120,220,20,10);
            g.drawOval(158,220,20,10);

            // Dibujamos un arco
            g.drawArc(128,235,40,35,190,160);

      }

      public static void main(String args[])
      {
            MiFrame mf = new MiFrame();
      }
}
```

Figura 16
Ejemplo completo con gráficos

8.3 Swing

Swing es el GUI que nos ofrece Java que consta e 9 packages que contienen centenares de clases e interfaces. Seguramente puede afirmarse sin duda a equivocarse que por el volumen de componentes que nos ofrece Java para crear nuestras aplicaciones gráficas, éste no tiene rival en el mercado.

La clase *javax.swing.Jcomponent* es la clase superior de toda la jerarquía de clases de *Swing*. Hereda de *java.awt.Container* y esta de *java.awt.Component* por lo que todo componete Swing es simultáneamente componente y contenedor.

El segundo concepto fundamental de Swing son los *layout managers*, que definen la forma de disponer los componentes por la pantalla.

El tercer y último concepto es el de evento. Un evento es un suceso asíncrono que se produce en nuestro programa y para el que hemos de prever (y programar) una reacción, por ejemplo cuando se pulsa la X de cerrar ventana (evento), hay que cerrar la misma (reacción).

8.3.1 Jerarquía de Swing

La jerarquía principal de Swing es la siguiente (a continuación explicaremos los más importantes en detalle):

1. JComponent
   - 1.1. AbstractButton
     - 1.1.1. JButton: típico botón
     - 1.1.2. JMenuItem
       - 1.1.2.1. JCheckBoxMenuItem: elemento dentro de una lista de menú que se puede seleccionar
       - 1.1.2.2. JMenu: cada una de las etiquetas que contiene un menú
       - 1.1.2.3. JRadioButtonMenuItem: elemento dentro de una lista de menú entre los cuales sólo se puede seleccionar uno simultáneamente
     - 1.1.3. JToggleButton: botón con dos estados posibles
       - 1.1.3.1. JCheckBox: elemento que puede estar seleccionado o no
       - 1.1.3.2. JRadioButton: usado con un ButtonGroup, sólo uno puede estar seleccionado
   - 1.2. JColorChooser: típico panel de selección de color
   - 1.3. JComboBox: lista desplegable de la cual se puede elegir un elemento
   - 1.4. JfileChooser: típico panel de selección de fichero
   - 1.5. JLabel: etiqueta, que no reacciona a eventos, donde se puede poner texto e imágenes
   - 1.6. JList: componente que contiene una lista de la cual podemos elegir uno o más componentes
   - 1.7. JMenuBar: barra superior del menú que contiene JMenu's
   - 1.8. JPanel: contendor genérico sobre el que se añaden componentes
   - 1.9. JPopUpMenu: menú emergente que aparece al clickar con el botón derecho del ratón
   - 1.10. JProgressBar: típica barra progreso de actividad no acabada

1.11. JScrollBar: barra se scroll
1.12. JScrollPane: panel con dos barras, vertical y horizanta, de scroll
1.13. JSeparator: línea separadora dentro de un menú
1.14. JSlider: típica barra de desplazamiento
1.15. JSplitPane: panel dividido en dos partes
1.16. JTabbedPane
1.17. JTableHeader
1.18. JTextComponent
    1.18.1. JEditorPane: facilita la creación de un editor
    1.18.2. JTextArea: área donde se puede introducir texto
    1.18.3. JTextField: ídem que el anterior pero de una sola línea
        1.18.3.1. JPasswordField: el texto se muestra con el símbolo que escojamos
1.19. JToolBar: la barra con iconos que suele aparecer en la parte superior
1.20. JToolTip: el texto emergente que aparece al situar el ratón sobre un componente y que aporta pistas sobre su uso
1.21. JTree: se correspondería a la parte izquierda del explorar de Windows

.

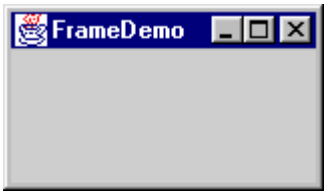
8.3.2 Índice visual

A continuación se muestra un "índice visual" de componentes extraído de *"The JavaTutorial"* que se puede encontrar on-line en http://java.sun.com/docs/books/tutorial/

8.3.2.1 Contenedores de alto nivel en la jerarquía

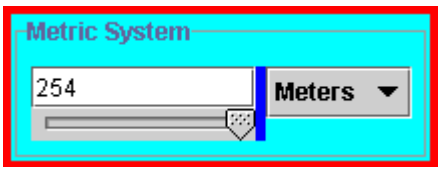
JApplet

JDialog



JFrame

8.3.2.2 Contenedores de propósito general

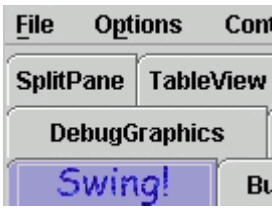


JPanel

JScrollPane

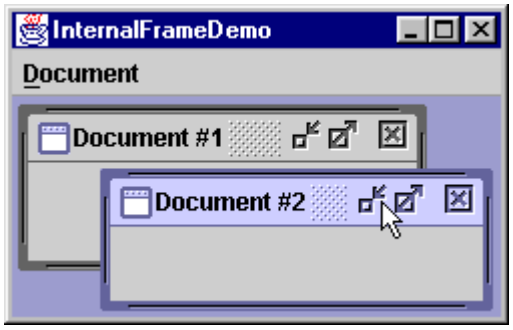
JSplitPane

JToolBar

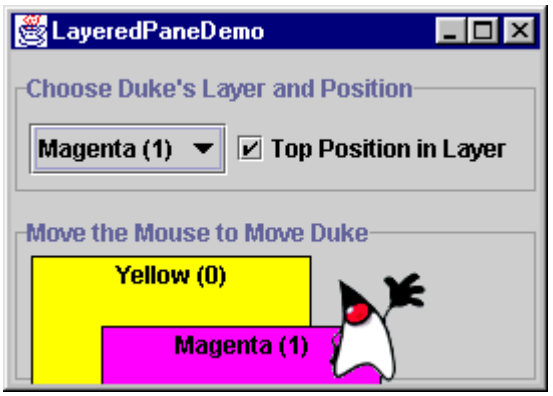


JTabbedPane

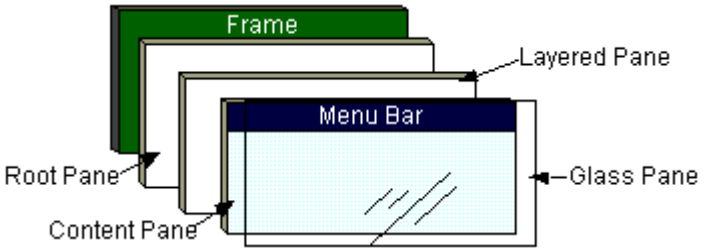
8.3.2.3 Contenedores de propósito especial



JInternalFrame

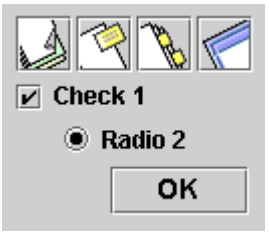


JLayeredPane

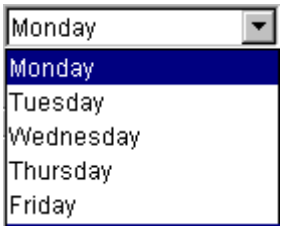


JRootPane

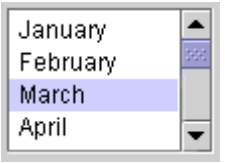
8.3.2.4 Controles básicos



Botones

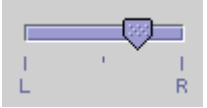


JComboBox



JList

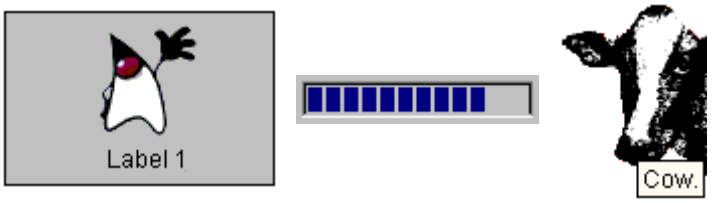
JMenu



JSlider

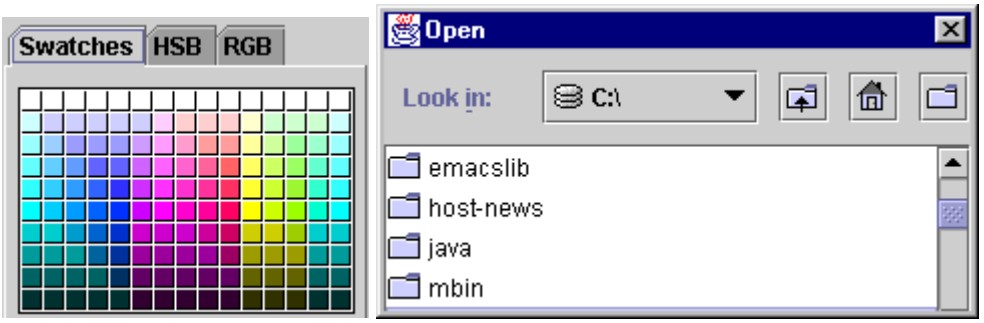
JTextField

8.3.2.5 Pantallas no editables

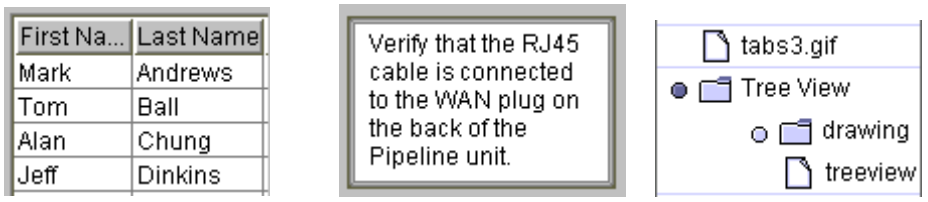


JLabel JProgressBar JToolTip

8.3.2.6 Pantallas editables o información formateada



JColorChooser JFileChooser



JTable JText JTree

8.3.3 Un ejemplo ilustrativo

A continuación os mostramos un primer ejemplo que usa componentes *Swing* y al cual nos iremos refiriendo a lo largo de los apartados siguientes. No os preocupéis si no entendéis alguna cosa del mismo en este momento.

```
01    import javax.swing.*;
02    import java.awt.*;
03
04    public class MiFrameConBotones extends JFrame
05    {
06          private JPanel pnlPrincipal;
07
08          private JButton btnOeste;
09          private JButton btnNorte;
10          private JButton btnEste;
11          private JButton btnSur;
12          private JButton btnCentro;
```

```
13
14          public MiFrameConBotones()
15          {
16                  super("Swing es divertido");
17                  setSize(500,500);
18
19                  pnlPrincipal = new JPanel();
20                  pnlPrincipal.setLayout(new  BorderLayout());
21
22                  btnOeste = new JButton("oeste");
23                  btnNorte = new JButton("norte");
24                  btnEste = new JButton("este");
25                  btnSur = new JButton("sur");
26                  btnCentro = new JButton("centro");
27
28                  pnlPrincipal.add(btnOeste,"West");
29                  pnlPrincipal.add(btnNorte,"North");
30                  pnlPrincipal.add(btnEste,"East");
31                  pnlPrincipal.add(btnSur,"South");
32                  pnlPrincipal.add(btnCentro,"Center");
33
34                  setContentPane(pnlPrincipal);
35
36                  setVisible(true);
37          }
38
39
40
41          public static void main(String args[])
42          {
43                  MiFrameConBotones  mf = new MiFrameConBotones();
44          }
45    }
```

8.3.4 Layouts

Como hemos visto en la introducción del apartado de Swing para diseñar una GUI tenemos que preocuparnos de qué componentes disponemos (botones, etiquetas, menús...) a qué eventos responden (como clickar un botón) y finalmente de cómo se distribuyen estos componentes.

Se puede hacer una división de todos los componentes en función de si son o no contenedores. Los contenedores son componentes que, valga la redundancia, pueden contener a otros. La manera en que estos se añaden (y la manera en la que se comporta el método add) depende del administrador de diseño (*LayoutManager*) que dicho contenedor tenga establecido. Una de los beneficios que aporta el uso de layouts es que cuando redimensionamos la ventana nuestra aplicación no se "desmonta" y los componentes se redistribuyen automáticamente respecto a como han sido situados sobre el administrador de diseño.

En nuestro ejemplo de referencia vemos como se añade un layout de tipo *BorderLayout* al *JPanel* pnlPrincipal en la línea 20.

Existen cinco tipos básicos de layouts: *FlowLayout, BorderLayout, CardLayout, GridLayout* y *GridBagLayout.* Pasemos a ver cada uno de estos tipos en detalle. Después veremos ejemplos de código a medida que vayamos explicando los diferentes componentes.

8.3.4.1 FlowLayout

Este es el *layout manager* más sencillo que vamos a encontrar y es el que asumen por defecto los *Applets* y los *JPanels*. El funcionamiento consiste en ir añadiendo los componentes línea a línea, cuando una línea esta llena se crea una nueva y se empieza a llenar.

Podemos usar tres constructores en esta clase:

FlowLayout()
FlowLayout(int alignment)
FlowLayout(int alignment, int hgap, int vgap)

El parámetro *alignment* es el alineamiento que queramos darle a elegir entre

```
FlowLayout.LEFT
FlowLayout.RIGHT
FlowLayout.CENTER
```

Los parámetros `hgap` y `vgap` definen la separación horizontal y vertical entre componentes.

8.3.4.2 BorderLayout

Este *layout manager* es el que usan los *JFrame's* por defecto. Divide la pantalla en cinco partes: North, South, East, West y Center, y nosotros especificamos en la llamada a add(...) en cual de ellas vamos a situar el componente.

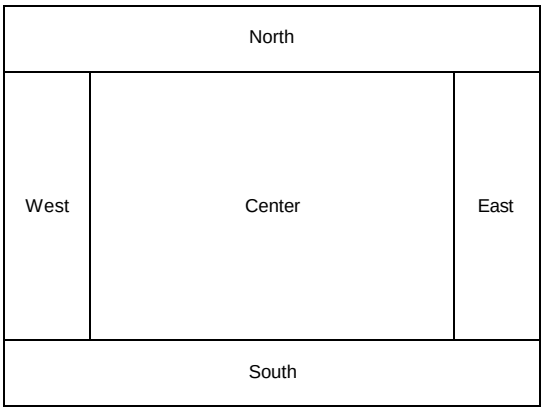


Figura 19
Estructura del *BorderLayout*

Los componentes situados en las zonas North y South se expanden todo el ancho necesario, posterio rmente las zonas East y West ocupan toda la altura que quede después de haber puesto los dos primeros componentes y la anchura que necesiten. Finalmente lo que situemos en la zona Center ocupará el resto del espacio disponible. Los constructores disponible s para esta clase son:

public BorderLayout()
public BorderLayout(int hgap, int vgap)

Las variables hgap y vgap definen la separación entre las zonas.

8.3.4.3 CardLayout

Este *layout manager* se basa en la idea de las pestañas. Sólo uno de los componentes definidos será mostrado simultáneamente y ocupará todo el espacio posible. Usualmente nos serviremos de algún tipo de eventos para pasar de un componente a otro. Como parece lógico, puede resultar conveniente en muchos casos conocer cual es el anterior y siguiente componente, es por ello que este *layout manager* nos ofrece algunos métodos propios:

public CardLayout()
public CardLayout(int hgap, int vgap)
void first(Container parent)
public void last(Container parent)
public void next(Container parent)
public void previous(Container parent)
public void show(Container parent, String name)

Es indispensable que conozcamos el Container padre del *layout.* Para añadir componentes a un Container que use este *layout manager* hemos de usar el método

```
void add(Component  o, String name)
```

El parámetro *name* identifica los componentes y es el mismo nombre que pasaremos como parámetro en el método show(...).

8.3.4.4 GridLayout

Este layout manager establece un sistema de filas y columnas permitiendo poner un componente dentro de cada casilla. Cuando añadimos los componentes a un contenedor gestionado por este layout no hemos de explicitar la posición donde queremos que aparezca, sino que iremos añadiendo componentes secuencialmente de izquierda a derecha y de arriba a abajo.

Otro aspecto a tener en cuenta en este manager de layout es que el componente que ocupe una celda variará su tamaño hasta llenar la totalidad del espacio disponible. Las celdas tendrán todas el mismo tamaño.

(0,0)	(1,0)	(2,0)
(1,0)	(1,1)	(2,1)
(2,0)	(2,1)	(2,2)

Figura 20
Ejemplo de *GridLayout*

Los constructores en este caso son:

public GridLayout(int rows, int cols)
public GridLayout(int rows, int cols, int hgap, int vgap)

8.3.4.5 GridBagLayout

Existe otro layout manager parecido a este pero más versátil que permite que un componente ocupe el espacio correspondiente a más de una celda: el GridBagLayout. No se explica en este manual debido a su complejidad, pero de todos modos, los valientes podéis consultar el manual de la API cuando quráis.

8.3.5 Componentes de Swing

Ahora vamos a estudiar cómo funcionan y qué métodos relevantes tienen los componentes más importantes. Si no os queda claro qué componente es alguno de ellos, remitíos al índice visual anterior.

8.3.5.1 JFrame

Ya hemos estado trabajando con esta clase anteriormente. Una *JFrame* es la clase que representa la ventana en la que se ejecutan las aplicaciones gráficas en Java. El procedimiento usual acostumbra a ser crear una clase nueva que herede de *JFrame*..

Normalmente los *JFrame's* se utilizan como contenedores primarios, es decir, no están contenidos dentro de otros contenedores.

A un frame se le suele poner un panel (explicados en el siguiente subapartado) principal a partir del cual se organizan los demás elementos. Para colocar este panel principal se usa el método

```
public void setContentPane(Cantainer  contentPane)
```

como se ve en la línea 34 de nuestro ejemplo.

8.3.5.2 JPanel

Un *JPanel* es una ventana invisible sobre las que se sitúan y organizan los elementos en la ventana. Podemos insertar cualquier tipo de componentes dentro de un panel, incluyendo otros paneles. Esta facilidad se hace imprescindible a la hora de crear interfaces gráficas complejas en las que haya que usar diferentes *Layout Managers* (también conocidos como administradores de diseño que veremos ahora enseguida).

El procedimiento habitual para poner un panel consiste en construirlo primero, añadir los componentes que se desee y luego se insertarlo en el contenedor adecuado.

Consta de más de 293 métodos heredados de *JComponent, Container* y *Component,* aparte de los heredados de *Object.* De entre todas estas operaciones, destacaremos las siguientes heredadas de *Container*:

public JPanel(): constructora;
public JPanel(LayoutManager layout): crea un nuevo *JPanel* con el administrados de diseño que se le pasa;
public Component add(Component comp): es el método usado para añdir componentes al panel. Devuelve el mismo componente que se le pasa;
public setLayout(Layout Manager l): establece el administrador de diseño dado;
public void remove(Component c): elimina el componente

En nuestro ejemplo podemos ver cómo funcionan algunos de estos métodos.

8.3.5.3 Canvas

Un *Canvas* es una "tela" rectangular en la que "pintamos" el texto o dibujos lo que queremos. La utilidad de los *canvas* está en que así nos evitamos tener que dibujar directamente en la ventana con las complicaciones que nos puede acarrear. Funciona exactamente como un *JPanel* (aunque no es un contenedor). Este objeto se puede utilizar para las técnicas de doble *buffering* para evitar los efectos de parpadeo, pintando primero sobre los *canvas* y luego añadiendo los *canvas* a la pantalla una vez que ya están construidos.

Una vez hemos dibujado el *Canvas* sólo hemos de añadirlo como cualquier otro componente. Es bastante usual derivar una clase de *Canvas* para redefinir las funciones que dibujen las formas que queremos y luego añadir este nuevo componente a la ventana o panel pertinente. El método en el que programaremos lo que queremos dibujar siempre será

```
public void paint(Graphics g)
```

8.3.5.4 JButton

Son componentes que generan un evento cuando clickeamos sobre ellos. Una de las principales características de los *JButton* es que admiten que su contenido sea texto y/o imágenes, cosa que hace que nuestras interfaces sean más atractivas. Algunos métodos a destacar son:

public JButton(String texto): crea un botón con el texto dado en su interior;
public JButton(Icon icono): crea un botón con el icono en su interior;
public String getLabel(): devuelve la etiqueta del botón;
public void setLabel(String label): establece una nueva etiqueta;
public void setEnabled(boolean b): si b es *false* el botón quedará deshabilitado y no podrá ser pulsado.

8.3.5.5 JLabel

Es un área que permite mostrar un String, una imagen o ambos. Una *JLabel* (etiqueta) no reacciona a los eventos del usuario por lo que no puede obtener el focus del teclado. Se suele utilizar en combinación con otros componentes que no disponen de la capacidad de mostrar su finalidad; por ejemplo, en combinado con un *JTextField* nos serviría para indicar al usuario qué esperamos que introduzca en el mismo. Algunos métodos importantes son:

public JLabel(String texto): crea una etiqueta con el texto dado;
public JLabel(Icon icono): la crea con un icono;
public JLabel(String text, int alineacionHorizontal): crea una etiqueta con un texto y una alineación dados. Ésta última puede ser LEFT, CENTER, RIGHT, LEADING o TRAILING definidas en el interfaz *SwingConstants*;
public void setText(String texto): establece la el texto que mostrará;
public String getText(): devuelve el texto;
public Icon getIcon(): devuelve el icono.

8.3.5.6 JTextField

El *JTextField* es un campo de texto de una sola línea. Hereda de la clase *javax.swing.text.JTextComponent.* Entre sus métodos destacamos los siguientes:

public JTextField(int numCols): crea un *JtextField* que ocupa numCols columnas. Si el texto que contiene sobrepasa este tamaño no se perderá aunque no se pueda mostrar todo simultáneamente;
public JTextField(String texot, int numCols): igual que el anterior pero además presentará un texto inicial;
public void setFont(Font fuente): establece la fuente con la que se mostrará el texto;
public String getText(): retorna su contentenido;
public String getSelectedText(): retorna el texto seleccionado en su interior;
public void copy(): copia el texto seleccionado al portapapeles de sistema;
public void cut(): igual que el anterior pero además elimina el texto seleccionado;
public void setEditable(boolean b): habilita o inhabilita la posibilidad de cambiar el texto contenido.

Llegados a este punto vamos a poner un nuevo ejemplo con los componentes vistos hasta ahora y con un *GridLayout*:

```
01    import javax.swing.*;
02    import java.awt.*;
03
04    public class Componentes1 extends JFrame
05    {
06          private JLabel lblIntroTexto;
07          private JTextField txtIntroTexto;
08          private JButton btnBorrar;
09          private JPanel pnlPrincipal;
10
11          public Componentes1()
12          {
13                super("Componentes1");
14
15                lblIntroTexto = new JLabel("Introduzca
texto",SwingConstants.CENTER);
16                txtIntroTexto = new JTextField(30);
17                btnBorrar = new JButton("Borrar");
18                pnlPrincipal = new JPanel();
19
20                pnlPrincipal.setLayout(new  GridLayout(3,1));
21                pnlPrincipal.add(lblIntroTexto);
22                pnlPrincipal.add(txtIntroTexto);
23                pnlPrincipal.add(btnBorrar);
24
25                setContentPane(pnlPrincipal);
26                setSize(200,200);
27                setVisible(true);
28          }
29
30          public static void main(String args[])
31          {
32                Componentes1  c = new Componentes1();
33          }
34    }
```

En las líneas de la 15 a la 18 creamos componentes, en la de la 20 establecemos el layout de la 21 a la 23 montamos los componentes en el panel y en la 25 insertamos el panel en nuestra ventana.

Figura 21
Aplicación Componentes1

8.3.5.7 JTextArea

Es un campo de texto que admite un nombre indeterminado de filas y columnas. Podemos escribir en él todo lo que queramos, ya que si nos pasamos de las dimensiones que muestra, aparecen barras de desplazamiento para poder mostrar toda el área de texto. Hereda de la clase *javax.swing.JTextComponent* igual que el *JTextField* por lo que, quitando las constructoras, también podremos aplicarle todos los métodos comentados para el *JTextField*. Otros métodos interesantes son:

public JTextArea(): crea una nueva área de texto con 0 filas y columnas;
public JTextArea(int filas, int columnas): la crea con el número de filas y columnas dados;
public JTextArea(String texto, int filas, int columnas): igual que el anterior pero con un texto inicial;
public int getRows(): devuelve el número de filas;
public int getColumns(): devuelve el número de columnas;
public void append(String texto): concatena el texto al final del texto ya existente en la *JTextArea*;
public void insert(String texto, int posicion): inserta el texto en la posición especificada

8.3.5.8 JCheckBox

Es un componente que puede estar seleccionado o no y que muestra su estado. Por convención puede estar seleccionado cualquier número de *JCheckBox*'es. Su apariencia es la de un cuadrado pequeño que está vacío cuando no está seleccionado y muestra una 'x'

cuando lo está. Además, puede mostrar un texto y/o imagen que indique su finalidad. Entre las operaciones de los *JCheckBox* destacamos:

public JCheckBox(): crea un *JCheckBox* sin texto ni icono y que no está seleccionado;
public JCheckBox(String texto, boolean seleccionado): lo crea con un texto asociado y un estado inicial;
public boolean isSelected(): devuelve *true* si está seleccionado;
public void setSelected(boolean b): lo selecciona si b es *true* y lo marca como no seleccionado si es *false*;
public String getText(): nos devuelve el texto asociado al *JCheckBox*. Esta operación reemplaza a *getLabel* que ha pasado a ser *deprecated*;
public void setEnabled(boolean b): permite habilitar o deshabilitar el componente para que puede ser utilizable o no por el usuario.

8.3.5.9 JRadioButton y BottonGroup

Muchas veces querremos poner una serie de casillas de verificación que estén agrupadas de manera que sólo puede seleccionarse una y que el hecho de marcar una implique desmarcar las otras. Esto lo podremos hacer gracias a la clase *JRadioButton* junto con la clase *ButtonGroup* (notad que en Swing no existe el equivalente al *CheckBoxGroup* de AWT). Métodos a destacar de *JRadioButton*:

public JRadioButton(): crea un *JRadioButton* que no está seleccionado;
public JRadioButton(String texto, boolean seleccionado): lo crea con un estado de selección inicial y un texto asociado;
public boolean isSelected()
public void setSelected(boolean b)
public String getText()
public void setEnabled(boolean b)

y los métodos a destacar del *ButtonGroup* son:

public ButtonGroup(): crea un nuevo *ButtonGroup*;
public void add(AbstractButton b): es posible añadir cualquier componente que herede de *AbstractButton*.

Si se añade más de un botón seleccionado, sólo el primero prevalecerá en este estado y los demás quedarán desactivados.

8.3.5.10 JComboBox

Es una lista desplegable de la que sólo se puede elegir una opción. Su equivalente AWT recibe el nombre de *Choice*. Destacamos las siguientes operaciones:

public JComboBox(): crea un *JComboBox* vacío;
public JComboBox(Object[] items): lo crear a partir de un array de objetos que contiene elementos;
public JComboBox(Vector items): obtiene los elementos del *Vector*;

public void addItem(Object item): añade un nuevo item al *JComboBox*;
public Object getItem(int indice): retorna el objeto que está situado en la posición indice;
public int getSelectedIndex(): devuelve el índice del item seleccionado;
public Object getSelectedItem(): devuelve el objeto seleccionado.

8.3.5.11 JList

La *JList* es una lista sobre la que podemos ver y seleccionar más de un elemento simultáneamente. En caso de que haya más elementos de los que se pueden visualizar simultáneamente a causa de las dimensiones de la lista, aparecerá una barra de desplazamiento vertical. Métodos relevantes:

public JList(): crea una *JList* vacía;
public JList(Object[] listaItems): la crea con los items contenidos en el array de objetos;
public JList(Vector listaItems): la crea con los elementos del *Vector*;
public int getSelectedIndex(): devuelve el índice del primer elemento seleccionado (-1 si no hay ninguno);
public int[] getSelectedIndices(): devuelve un array con los índices que están seleccionados;
public Object getSelectedValue(): da la referencia al objeto seleccionado;
public Object[] getSelectedValues(): devuelve un array con los *Object* seleccionados.

Os pondremos ahora otro ejemplo más complejo donde se usan muchos de los componentes vistos hasta ahora en combinación con paneles con distintos layouts:

```
01    import javax.swing.*;
02    import java.awt.*;
03
04    public class Componentes2 extends JFrame
05    {
06          private JCheckBox chkJava;
07          private JCheckBox chkProfes;
08          private JCheckBox chkJedi;
09          private JPanel pnlChecks;
10
11          private JLabel lblLenguajes;
12          private JList lstLenguajes;
13          private JPanel pnlLenguajesPrimario;
14          private JPanel pnlLenguajesSecundario;
15          private String[]lenguajes;
16
17          private JRadioButton rbtMH;
18          private JRadioButton rbtSobresaliente;
19          private JRadioButton rbtNotable;
20          private JRadioButton rbtOtras;
21          private ButtonGroup grpNotas;
22          private JLabel lblNotas;
23          private JPanel pnlNotasTerciario;
24          private JPanel pnlNotasSecundario;
25          private JPanel pnlNotasPrimario;
26
27          private JPanel pnlPrincipal;
```

```
28
29          public Componentes2()
30          {
31                super("Componentes2");
32
33                chkJava = new JCheckBox("¡Me gusta Java!",true);
34                chkProfes = new JCheckBox("Los profes de Java son
geniales",true);
35                chkJedi = new JCheckBox("Jedi es genial",true);
36                pnlChecks = new JPanel(new FlowLayout());
37                pnlChecks.add(chkJava);
38                pnlChecks.add(chkProfes);
39                pnlChecks.add(chkJedi);
40
41                lblLenguajes = new JLabel("Seleccione lenguajes de
programación que domine",SwingConstants.CENTER);
42                lenguajes = new String[9];
43                lenguajes[0]="Java";
44                lenguajes[1]="C++";
45                lenguajes[2]="Eiffel";
46                lenguajes[3]="Pascal";
47                lenguajes[4]="Perl";
48                lenguajes[5]="Clips";
49                lenguajes[6]="Cobol";
50                lenguajes[7]="Ada";
51                lenguajes[8]="PHP";
52                lstLenguajes = new JList(lenguajes);
53                pnlLenguajesPrimario  = new JPanel(new
GridLayout(2,1));
54                pnlLenguajesPrimario.add(lblLenguajes);
55                pnlLenguajesSecundario  = new JPanel(new
FlowLayout());
56                pnlLenguajesSecundario.add(lstLenguajes);
57                pnlLenguajesPrimario.add(pnlLenguajesSecundario);
58
59                rbtMH = new JRadioButton("Matricula de Honor",true);
60                rbtSobresaliente  = new
JRadioButton("Sobresaliente",true);
61                rbtNotable = new JRadioButton("Notable",false);
62                rbtOtras = new JRadioButton("Otras",false);
63                grpNotas = new ButtonGroup();
64                grpNotas.add(rbtMH);
65                grpNotas.add(rbtSobresaliente);
66                grpNotas.add(rbtNotable);
67                grpNotas.add(rbtOtras);
68                pnlNotasTerciario  = new JPanel(new GridLayout(2,2));
69                pnlNotasTerciario.add(rbtMH);
70                pnlNotasTerciario.add(rbtSobresaliente);
71                pnlNotasTerciario.add(rbtNotable);
72                pnlNotasTerciario.add(rbtOtras);
73                lblNotas = new JLabel("¿Qué nota vas a sacar en el
examen de Java?",SwingConstants.CENTER);
74                pnlNotasSecundario  = new JPanel(new FlowLayout());
75                pnlNotasSecundario.add(pnlNotasTerciario);
76                pnlNotasPrimario  = new JPanel(new BorderLayout());
77                pnlNotasPrimario.add(pnlNotasSecundario,"South");
78                pnlNotasPrimario.add(lblNotas,"North");

79
80                pnlPrincipal  = new JPanel(new BorderLayout());
81                pnlPrincipal.add(pnlChecks,"North");
```

```
82                pnlPrincipal.add(pnlLenguajesPrimario,"Center");
83                pnlPrincipal.add(pnlNotasPrimario,"South");
84                pnlPrincipal.add(new JButton("boton
izquierdo"),"West");
85                pnlPrincipal.add(new JButton("boton
derecho"),"East");
86
87                setContentPane(pnlPrincipal);
88                setSize(600,500);
89                setVisible(true);
90          }
91
92          public static void main(String args[])
93          {
94                Componentes2 c = new Componentes2();
95          }
96    }
```

Figura 22
Aplicación Componentes2

8.3.5.12 Menús

Entendemos por menú toda estructura superior de una aplicación gráfica que nos permite desplegar unas listas de operaciones que a su vez pueden contener otras listas de operaciones. La estructura básica de un menú se compone de *JMenuBar* (la barra superior) de la que cuelgan los menús (*JMenu*) a los que añadimos elementos (*JMenuItem*) que pueden ser *JCheckBoxMenuItem, JRadioButtonMenuItem* e incluso *JMenu*. Veamos ahora los métodos de cada una de estas clases:

JMenuBar:

public JMenuBar(): crea una nueva *JMenuBar*;
public JMenu add(JMenu m): añade un *JMenu* a la *MenuBar*;
public JMenu getMenu(int indice): devuelve el menú asociado al índice correspondiente.

JMenuItem (superclase de *JCheckBoxMenuItem, JradioButtonItem* y *JMenu*):

public JMenuItem(): crea un elemento de menú vacío;
public JMenuItem(String texto): lo crea con un string asociado;
public JMenuItem(Icon icono): es creado con una imagen asociada que será la que se muestre en el menú;
public void setEnabled(boolean b): indicamos si queremos que se pueda seleccionar o no.

JMenu:

public JMenu(String s): crea un nuevo *JMenu* que mostrará el string especificado;
public JMenu add(JMenuItem menuItem): añade un JMenuItem al JMenu;
public JMenu add(String s): crea un nuevo elemento de menú con el texto especificado y lo pone al final de este *JMenu*;
public void addSeparator(): añade una línea horizontal separadora;
public JMenuItem getItem(int pos): devuelve el *JMenuItem* que contiene la posición indicada

A continuación se muestra un ejemplo sencillo:

```
01    import javax.swing.*;
02    import java.awt.*;
03
04    public class PrimerMenu extends JFrame
05    {
06          public PrimerMenu()
07          {
08                super("Menús");
09
10                JMenuBar mb = new JMenuBar();
11                JMenu m1 = new JMenu("Archivo");
12                JMenu m2 = new JMenu("Edición");
13                JMenuItem mi11 = new JMenuItem("Abrir");
14                JMenuItem mi12 = new JMenuItem("Cerrar");
15                JMenuItem mi13 = new JMenuItem("Guardar");
```

```
16                JMenuItem mi14 = new JMenuItem("Imprimir");
17                JMenuItem mi21 = new JMenuItem("Cortar");
18                JMenuItem mi22 = new JMenuItem("Pegar");
19                m1.add(mi11);
20                m1.add(mi12);
21                m1.add(mi13);
22                m1.addSeparator();
23                m1.add(mi14);
24                m2.add(mi21);
25                m2.add(mi22);
26                mb.add(m1);
27                mb.add(m2);
28                setJMenuBar(mb);
29                setSize(400,300);
30                setVisible(true);
31          }
32
33          public static void main(String args[])
34          {
35                PrimerMenu pm = new PrimerMenu();
36          }
37    }
38
39
```

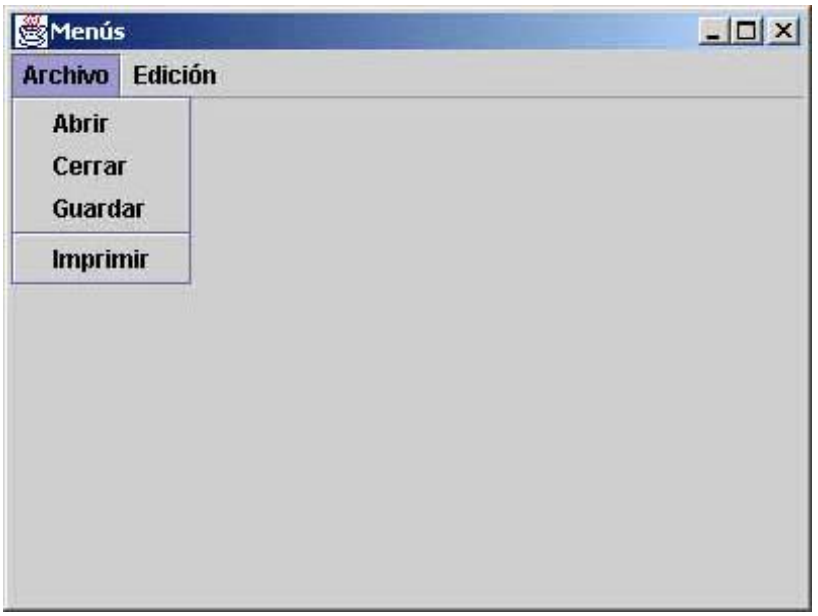


Figura 23
Aplicación PrimerMenu

8.3.5.13 JDialog

Un *JDialog* es una ventana que tiene propietario. La diferencia principal sobre el *JFrame* radica en que el *JDialog* puede ser modal. El hecho de ser modal significa que el usuario no puede cerrar la ventana ni cambiar el focus a otra ventana de nuestra aplicación hasta que el propio *JDialog* lo decida. Un *JDialog* modal sería la clásica ventana de Windows que dice: "Esta aplicación ha realizado una operación no válida y se cerrará". Veamos los métodos más interesantes:

public JDialog(): crea un *JDialog* no modal y sin padre;
public JDialog(Frame propietario, String titulo, boolean modal): crea un *JDialog* con un título, un padre y una modalidad determinada;
public void setLayout(Layout Manager): establece el administrador de diseño del *JDialog*;
public Container getContentPane(): obtiene el panel principal sobre el cual añadiremos nuestros componentes;
public boolean isModal(): devuelve un booleano indicando si es modal o no;
public void setModal(boolean b): establece la modalidad

9 EVENTOS

Ahora ya conocemos los diferentes componentes que existen en la GUI de Java y sabemos como distribuirlos y diseñarlos pero aún no hemos hecho que interactúen con el usuario: si nosotros apretábamos sobre un botón la aplicación se mostraba totalmente estática y no teníamos manera de gestionar estos *eventos*.

9.1 Conceptos básicos

Cuando el usuario realiza una acción a nivel del interfaz gráfico (pulsa una tecla o un botón del ratón), se produce un evento que puede ser tratado. Los eventos son objetos que describen lo que sucede. Existen una gran variedad de clases de eventos para describir las diferentes categorías de acciones de usuario.

Un event source es el **generador de un evento**. Por ejemplo, la pulsación de un componente *JButton* con el ratón genera un evento de tipo *ActionEvent* con el botón como *source*. La instancia *ActionEvent* es un objeto que contiene información sobre los eventos que acaban de ocurrir. En él están accesibles:

public getActionCommand(): retorna el nombre del comando asociado con la acción.

public getModifiers(): retorna los modificadores capturados durante la acción que ha generado el evento.

Un **gestor de eventos**, o event handler, es un método que recibe un objeto evento (cualquier clase que herede de *Event*), lo descifra y procesa la interacción del usuario.

El modelo de eventos de Java 2 (y desde la versión 1.1) recibe el nombre de **modelo de delegación** en contraposición al modelo jerárquico de versiones anteriores. En este manual no veremos el modelo jerárquico por ser obsoleto y estudiaremos directamente el modelo de delegación.

El modelo de delegación de eventos apareció junto con el JDK 1.1. En este modelo, los eventos se envían al componente que los ha originado, pero es cuestión de cada componente tener registradas una o más clases llamadas escuchas *(listeners*), que contienen gestores de eventos que pueden recibir y procesar un evento. De este modo el gestor de eventos puede estar en un objeto totalmente separado del componente. Las escuchas son clases que implementan el interfaz *Listener* apropiado.

Los eventos son objetos que únicamente se envían a los *listeners* registrados en el componente. Cada evento tiene un interfaz de escucha correspondiente que define qué métodos deben definirse en una clase que quiera recibir y tratar el evento correspondiente. La clase que implementa el interfaz debe definir dichos métodos y entonces se puede registrar como un *listener*.

Los eventos producidos sobre componentes que no tienen escuchas registradas no se propagan. Todo esto parece ser muy enrevesado, pero ya veréis como con un pequeño ejemplo todo queda clarificado. Vamos a crear una ventana simple con un botón:

```
01    import javax.swing.*;
02    import java.awt.*;
03
04    public class Pulsame extends JFrame
05    {
06          private JButton btnCambiarColor;
07          private Container pnlPrincipal;
08
09          public Pulsame()
10          {
11                super("Los eventos son fáciles");
12
13                btnCambiarColor = new JButton("¡Púlsame!");
14                btnCambiarColor.addActionListener(new  GestorBoton());
15                pnlPrincipal = getContentPane();
16                pnlPrincipal.add(btnCambiarColor);
17
18                setSize(100,100);
19                setVisible(true);
20          }
21
22          public static void main(String args[])
23          {
24                Pulsame cc = new Pulsame();
25          }
26    }
```

Lo único que no conocemos de este código es la línea 14. En esa línea se le está asociando al botón un gestor de eventos. Por tratarse de un botón se le tiene que añadir uno gestor de tipo *ActionListener*. Cuando alguien pulse el botón el evento generado ira a para a esa instancia de *GestorBoton* que se le ha asociado. Veamos ahora el código de *GestorBoton*:

```
01    import java.awt.event.*;
02
03    public class GestorBoton implements ActionListener
04    {
05          private int contador=0;
06
07          public void actionPerformed(ActionEvent  e)
08          {
09                System.out.println("El  botón ha sido pulsado
"+contador+" veces\r\n");
10                contador++;
11          }
12    }
```

Vayamos poco a poco. Lo primero que vemos es que nuestra clase implementa la interfaz *ActionListener* (línea 3). Si no fuera así no sería un gestor válido para este tipo de eventos. Recordemos que cuando una clase implementa una interfaz debe dar código para todos los métodos que esta define. Si nos vamos a la documentación de la API para este interfaz vemos que sólo define un método *public void actionPerformed(ActionEvente e)* que recibe el evento: este es el método que se llama cuando se produce una pulsación sobre el botón y es ahí donde escribimos el código que queremos que se ejecute. Fijaos que nuestra

clase además podría tener todos los métodos (incluso constructores si fuera necesario) y atributos que necesitara.

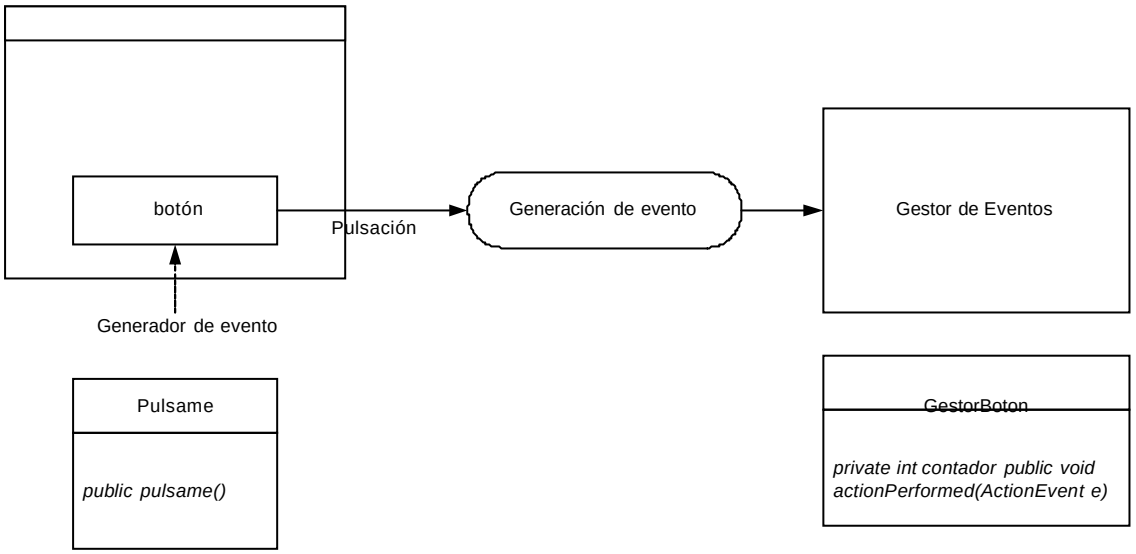


Figura 24
Gestión de eventos

> **Ejercicio 9.1**
>
> A partir de la clase *Componentes1* de la página 86 hacer que al apretar el botón se borre el contenido del *JTextField* y que el salga por la consola. No uséis una clase aparte para implementar el gestor de eventos.

9.2 Interfaces

Veamos ahora detalladamente qué tipos de interfaces hay para cada tipo de evento que podemos tratar.

9.2.1 Interfaz *ActionListener*

Implementaremos esta interfaz cuando queramos detectar eventos de tipo *ActionEvent.* Estos eventos se producen por ejemplo cuando se pulsa un botón o se pulsa *return* sobre un *JTextArea*, también cuando se selecciona un ítem de una lista o de un menú.

Para poder detectar estos eventos deberemos implementar la interfaz *ActionListener* e implementar el método

```
public void actionPerformed(ActionEvent  e)
```

Para hacer que un objeto pueda escuchar este tipo de eventos de otro objeto, al segundo le tendremos que invocar el método

```
public addActionListener(ActionListener)
```

9.2.2 Interfaz *ItemListener*

Implementaremos esta interfaz para escuchar los eventos que pueden producir los objetos *JCheckbox*, *JList* y *JCheckboxMenuItem.* El método que deberemos implementar es:

```
public void itemStateChanged(ItemEvent e)
```

Para añadir una escucha a un objeto de los antes mencionados le deberemos invocar el método

```
addItemListener(ItemListener)
```

9.2.3 Interfaz *WindowListener*

Implementaremos esta interfaz cuando queramos definir un objeto con capacidad para escuchar eventos de tipo *WindowEvent.*

Los productores de estos eventos son los objetos de tipo *JDialog* y *JFrame*. Podemos darnos de alta como *WindowListener* con la llamada al método

```
public void addWindowListener(WindowListener w)
```

Al implementar esta interfaz deberemos implementar los métodos siguientes:

public void windowOpened(WindowEvent e): detecta que la ventana ha sido abierta;
public void windowActivated(WindowEvent e): detecta que la ventana ha sido activada;
public void windowDeactivated(WindowEvent e): detecta que la ventana ha sido desactivada;
public void windowIconified(WindowEvent e): detecta que la ventana ha sido iconificada (minimizada);
public void windowClosed(WindowEvent e): detecta que la ventana ha sido cerrada (este evento se produce cuando se hace una llamada al método *dispose()* sobre la ventana;
public void windowDeiconified(WindowEvent e): detecta que la ventana ha sido desiconificada (maximizada o restaurada);
public void windowClosing(WindowEvent e): detecta que se ha pulsado el botón de cerrar la ventana.

Tened en cuenta que deberéis dar código para todos los métodos de la interfaz para poder compilar vuestro controlador. Esto puede ser un poco engorroso cuando queremos implementar por ejemplo, sólo el tratamiento de cerrar la ventana. Existe una solución para este "problema": existe una clase que se llama *WindowAdpater* que implementa *WindowListener* y que tiene "definidos" todos estos métodos. En realidad el código que da para cada uno de ellos es hacer nada, pero si a la hora de implementar nuestro gestor de

ventanas heredamos de esta clase de cara al compilador ya es suficiente y sólo tendremos que redefinir aquellos métodos que de verdad nos interesen.

9.2.4 Interfaz *ComponentListener*

Para marcar un objeto con la capacidad de escuchar eventos de tipo *ComponentEvent* deberemos implementar la interfaz *ComponentListener*. Los objetos que producen estos eventos son los *JDialog* y los *JFrames*.

Para añadir a un productor de eventos un oyente de sus eventos invocaremos el método:

```
public void addComponentListener(ComponentListener  c)
```

Los métodos que debemos definir al implementar esta interfaz son:

public void componentResized(ComponentEvent e)
public void componentMoved(ComponentEvent e)
public void componentShown(ComponentEvent e)
public void componentHidden(ComponentEvent e)

9.2.5 Interfaz *AdjustmentListener*

Para escuchar los eventos de tipo *AdjustmentEvent* producidos por la clase *JScrollbar* deberemos implementar la interfaz *AdjustmentListener*.

Para darnos de alta como *AdjustmentListeners* deberemos invocar la operación:

```
public void addAdjustmentListener(AdjustementListener  adj)
```

El método que debemos implementar para capturar el evento es

```
public void adjustmentValueChaged(AdjustementEvent  e)
```

9.2.6 Interfaz *MouseListener*

Como su nombre indica esta interfaz nos permitirá escuchar los eventos de tipo *MouseEvent*, que son los producidos por el ratón al interaccionar con cualquiera de las clases *Canvas*, *JDialog*, *JFrame* , *JPanel* y *JWindow.*

Para añadir a cualquiera de estas clases un oyente de eventos deberemos invocar al método

```
public void addMouseListener(MouseListener  ml)
```

Los métodos que vienen definidos por esta interfaz son:

public void mouseClicked(mouseEvent e)
public void mousePresed(mouseEvent e)
public void mouseReleased(mouseEvent e)

public void mouseEntered(mouseEvent e)
public void mouseExited(mouseEvent e)

9.2.7 Interfaz *MouseMotionListener*

Cualquiera de los objetos de las clases *Canvas, JDialog, JFrame, JPanel* y *JWindow* pueden producir eventos de este tipo.

Para marcar que un objeto puede escuchar este tipo de eventos deberá implementar la interfaz *MouseMotionListener*, y deberemos haberlo añadido al productor de eventos mediante el método:

```
public void addMouseMotionListener(MouseMotionListener  ml)
```

Los métodos de esta interfaz son:

public void mouseDragged(MouseEvent e)
public void mouseMoved(MouseEvent e)

9.2.8 Interfaz *FocusListener*

Cuando un componente recibe el foco, es decir, es el elemento de la pantalla que está activo se producen eventos de tipo *FocusEvent.*

Para hacer que un objeto pueda escuchar eventos de tipo *FocusEvent* deberá implementar la interfaz *FocusListener*, y además deberá ser añadido como *FocusListener* con el método

```
public void addFocusListener(FocusListener  fl)
```

Los métodos de esta interfaz son:

public void focusGained(FocusEvent e)
public void foculsLost(FocusEvent e)

9.2.9 Interfaz *KeyListener*

Hasta ahora hemos visto como interactuar con el programa a través de los evententos generados cuando el usuario manipula componentes de la interfaz gráfica, pero otra manera muy habitual de interactuar con el software es a través del teclado. Con la interfaz *KeyListener* seremos capaces de detectar y tratar los eventos generados por la pulsación de una tecla o por combinaciones de ellas, los *KeyEvent.*

Los gestores que implementan esta interfaz sólo pueden añadirse a aquellos componentes de la GUI que son capaces de recibir el foco del teclado.

Podemos detectar dos tipos básicos de eventos: la *pulsación* (entendiendo por pulsación el hecho de presionar la tecla y de liberarla a continuación) de una tecla que dé como resultado un carácter UNICODE (por ejemplo la pulsación de cualquier tecla alfanumérica) o el hecho de *presionar* y *soltar* una tecla (o conjunto de ellas). Los primeros

eventos, que son totalmente independientes de la plataforma reciben el nombre de *key-typed* los segundos, que pueden tener dependencia tecnológica, se llaman respectivamente *key-pressed* y *key-released*.

En general deberíamos poder trabajar siempre con los eventos de tipo *key-typed*, pero si quisieramos detactar la pulsación de teclas que no generan ningún carácter UNICODE tales como las teclas de función, control, etc, no tenemos más remedio que gestionar los eventos del segundo tipo.

La interfaz *KeyListener* define tres métodos que deb en ser redefinidos en el gestor que la implemente y que se corresponden a cada uno de los eventos que hemos visto:

public void keyTyped(KeyEvent e)
public void keyPressed(KeyEventt e)
public void keyReleased(KeyEvent e)

Veamos ahora un pequeño ejemplo que aclare todos estos conceptos. Este ejemplo está inspirado en el que aparece en el *Java Tutorial* (sólo en el resultado final y no en la implementación, que me parece que yo he conseguido que sea más didáctica). Este ejemplo muestra un *JTextField* editable y un *JTextArea* no editable que va mostrando la información de todos los eventos del teclado que se generan sobre el campo de texto. Este ejemplo es interesante por dos motivos: por un lado vemos como se utilizan los y tratan este tipo de eventos y por otro lado el programa en sí mismo es bastante pedagógico porque ayuda a comprender como se generan estos eventos. Antes de que os lo comente, compiladlo y experimentad un poco.

```
001 import java.awt.*;
002 import java.awt.event.*;
003 import javax.swing.*;
004
005 public class VisorEventosTeclado extends JFrame
006         implements KeyListener, ActionListener
007 {
008     private int numeroEvento = 1;
009
010     private JTextField txtEntrada;
011     private JTextArea txtInfo;
012     private JButton btnBorrar;
013
014     private JScrollPane pnlInfo;
015     private JPanel pnlPrincipal;
016
017     public VisorEventosTeclado()
018     {
019         super("Visor Eventos Teclado");
020
021         // Construimos la interfaz
022         txtEntrada = new JTextField();
023         txtEntrada.addKeyListener(this);
024
025         txtInfo = new JTextArea();
026         txtInfo.setEditable(false);
027         // Ponemos el panel de información en un ScrollPanel
028         pnlInfo = new JScrollPane(txtInfo);
029
```

```
030         btnBorrar = new JButton("Borrar");
031         btnBorrar.addActionListener(this);
032 
033         pnlPrincipal = new JPanel();
034         pnlPrincipal.setLayout(new  BorderLayout());
035         pnlPrincipal.add(txtEntrada,BorderLayout.NORTH);
036         pnlPrincipal.add(pnlInfo,BorderLayout.CENTER);
037         pnlPrincipal.add(btnBorrar,BorderLayout.SOUTH);
038 
039         setContentPane(pnlPrincipal);
040 
041         addWindowListener(new  GestorVentana());
042         setSize(400,600);
043         setVisible(true);
044     }
045 
046 
047     // Gestión de la pulsación del botón
048     public void actionPerformed(ActionEvent  e)
049     {
050         txtEntrada.setText("");
051         txtInfo.setText("");
052     }
053 
054     // Gestión del teclado
055     public void keyTyped(KeyEvent e)
056     {
057         mostrarInfo(e,"keyTyed");
058     }
059 
060     public void keyPressed(KeyEvent e)
061     {
062         mostrarInfo(e,"keyPressed");
063     }
064 
065     public void keyReleased(KeyEvent e)
066     {
067         mostrarInfo(e,"keyReleased");
068     }
069 
070     private void mostrarInfo(KeyEvent  e, String tipo)
071     {
072         char c = e.getKeyChar();
073         int keyCode = e.getKeyCode();
074         int modifiers = e.getModifiers();
075 
076         String charString;
077         if(Character.isISOControl(c))
078         // miramos si no es un carácter imprimible
079             charString = "\tCarácter: (no imprimible)\n";
080         else
081             charString = "\tCarácter: '"+ c + "'\n";
082 
083         String keyCodeString = "\tCódigo: " + keyCode +
084                 " (" + KeyEvent.getKeyText(keyCode)  + ")\n";
085 
086         String modString = "\tModificadores: " + modifiers;
087         String tmpString=KeyEvent.getKeyModifiersText(modifiers);
088         if (tmpString.length()>0)
089             modString += " (" + tmpString + ")\n\n";
090         else
```

```
091             modString += " (sin modificadores)\n\n";
092
093
094         String nuevoString = "Evento numero " + numeroEvento +
095                              " de tipo " + tipo + " detectado:\n" +
096                              charString + keyCodeString + modString;
097
098         txtInfo.append(nuevoString);
099         numeroEvento++;
100     }
101
102     public static void main(String[]args)
103     {
104         VisorEventosTeclado vet = new VisorEventosTeclado();
105     }
106 }
```

Si habéis hecho alguna prueba, podéis comprobara que siempre que pulséis alguna tecla que se corresponde a algún carácter UNICODE (y por tanto imprimible) se generan los tres eventos, pero cuando pulsáis alguna tecla que no tiene una representación sólo se generan los eventos de presioar y liberar tecla.

Fijaos también que los modificadores contienen información sobre qué teclas adicionales pueden estar presionadas en un momento dado permitiendo detectar, por ejemplo, que hemos pulsado A + CTRL o C + CTRL + ALT.

He resaltado en negrita aquellas partes del código que resultan interesantes para lo que ahora estamos tratando:

en la línea 6 vemos que nuestra clase implemente la interfaz *KeyListener* y por tanto seremos nosotros mismos el gestor que trate los eventos del teclado;
en la línea 23 podemos ver como se añade el gestor al campo de texto. Esto significa que los eventos de tipo teclado sólo serán generados y tratados cuando el foco esté sobre este componente;
como nuestra clase es el gestor en las líneas 55, 60 y 65 tenemos sobrescritos los métodos correspondientes a la interfaz, que básicamente se encargan de delegar en otro método la impresión de la información correspondiente.

El método *mostrarInfo* que es el encargado de imprimir la información generada por el evento es interesantes porque permite ver algunas de los métodos consultores que tiene la clase *KeyEvent.* También los he resaltado en negrita.

9.2.10 Tabla resumen de interfaces

Tabla 7. Resumen de listeners

Interfaz	Componentes	Gestor de evento	Métodos que redefine la interfaz
ActionListener	*JButton* *JTextField* *JTextArea* *JComboBox*	*ActionListener*	*actionPerformed(ActionEven e)*
ItemListener	*JCheckBox JList* *JCheckBoxMenuItem*	*ItemListener*	*itemStateChanged(ItemEvent e)*
ChangeListener	*JSlider*	*ChangeListener*	*stateChanged(ChangeEvent e)*
ComponentListener	*JDialog*	*ComponentListener*	*componentHidden(ComponentEvent e)*
	JFrame		*componentMoved(ComponentEvent e)* *componentResized(ComponentEvent e)* *componentShown(ComponentEvent e)*
AdjustmentListener	*JScrollBar*	*AdjustementListener*	*adjustmentValueChanged(AdjustmentEvent e)*
MouseListener	*Canvas*	*MouseListener*	*mouseClicked(MouseEvent e)*
	JDialog *JFrame* *JPanel* *JWindow*		*mouseEntered(MouseEvent e)* *mouseExited(MouseEvent e)* *mousePressed(MouseEvent e)* *mouseReleased(MouseEvent e)*
MouseMotionListener	*Canvas*	*MouseMotionListener*	*mouseDragged(MouseEvent e)*
	JDialog *JFrame* *JPanel* *JWindow*		*mouseMoved(MouseEvent e)*
FocusListener	Cualquier componente	*FocusListener*	*focusGained(FocusEvent e)*
	con capacidad de obtener el foco		*focusLost(FocusEvent e)*
WindowListener	*JDialog*	*WindowListener*	*windowActivated(WindowEvent e)*
	JFrame		*windowClosed(WindowEvent e)* *windowClosing(WindowEvent e)* *windowDeactivated(WindowEvent e)* *windowDeiconified(WindowEvent e)* *windowIconified(WindowEvent e)* *windowOpened(WindowEvent e)*

Ejercicio 9.2

Poner en un *JPanel* tres botones: “amarillo”, “azul” y “rojo”. Programar los eventos necesarios para que cada vez que clickemos cualquiera de los botones se cambie el color del fondo del panel. Programad también el evento cerrar ventana.

Ejercicio 9.3

Haced lo mismo pero en vez de botones utilizad *JSlider*s (ayudaos de la información de la tabla anterior y de la documentación de la API). Utilizad una para cada color primario (rojo, verde y azul) que tengan un rango 0-255 y que. Cada vez que movamos una de las barras tiene que actualizarse el color de fondo.

9.3 Look and Feel

El *“look and feel”* o en español *“aspecto y tacto adaptable”* es una característica que permite a los desarrolladores construir aplicaciones que se ejecutarán en cualquier plataforma como si se hubiesen diseñado y construido específicamente para ellas. Un programa ejecutado en el entorno Windows tendrá el aspecto y el comportamiento como si hubiese sido desarrollado para ese entorno; el mismo programa ejecutado en la plataforma UNIX se comportará como si se hubiese desarrollado para esa plataforma.

Los desarrolladores pueden crear sus propios componentes Swing con el aspecto y funcionalidad que deseen. Esto incrementa la fiabilidad y la coherencia de las aplicaciones y applets desplegados a través de diferentes plataformas. Por ejemplo, el código en negrita del siguiente ejemplo (que es una modificación del método *main* del r ejemplo *Componentes2* del capítulo anterior en la página 89) hace que se configure el *look and feel* de la máquina donde se está ejecutando la aplicación Java:

```
public static void main(String args[])
{
      try
      {

      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName
());
      }
      catch(Exception e)
      {
            System.out.println("No establecer el Look&Feel”);
      }

      Componentes2 c = new Componentes2();
}
```

Como yo estoy trabajando en una máquina con entorno *Windows* obtengo un aspecto tal como el que se muestra en la figura 25.

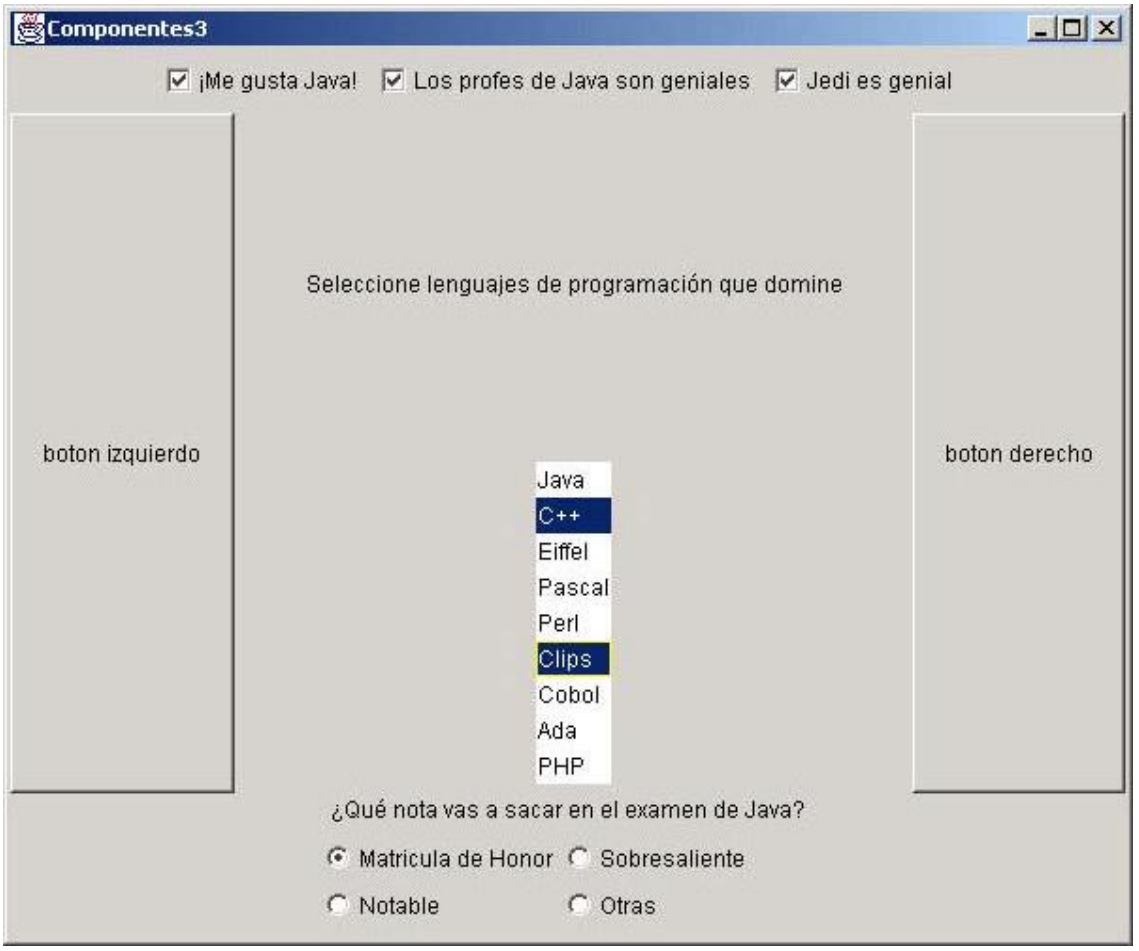


Figura 25
Aplicación Componentes3

En este ejemplo hemos elegido el *look and feel* nativo del host donde se ejecuta la máquina pero podemos elegir muchos más pasándole el parámetro a decuado al método *UIManager.setLookAndFeel(...)*. Por ejemplo:

UIManager.getCrossPlatformLookAndFeelClassName(): se devuelve el *Java look & Feel;*
UIManager.getSystemLookAndFeelClassName(): devuelve, como hemos visto en el ejemplo, el *look and feel* correspondiente a la plataforma donde se ejecuta la aplicación Java;
"javax.swing.plaf.metal.MetalLookAndFeel";
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel": se corresponde con el *look and feel* de Windows que actualmente sólo puede usarse en esa plataforma;
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
"javax.swing.plaf.mac.MacLookAndFeel": especifica el *Mac OS Look & Feel* que sólo puede ser usado en plataformas Mac.

Si nos fijamos en este ejemplo el *look & feel* se establece justo antes de construir la aplicación gráfica; de todos modos, también podemos cambiarlo una vez ya se ha mostrado. Veremos cómo se hace a través de un ejemplo completo:

```
01 import javax.swing.*;
02 import java.awt.*;
03 import java.awt.event.*;
04
05 public class CambiarLook extends JFrame implements
06                WindowListener, ActionListener
07 {
08    private JButton btnJava;
09    private JButton btnHost;
10    private JTextField txtInfo;
11    private JPanel pnlPrincipal;
12
13    public CambiarLook()
14    {
15          super("Cambiar Look And Feel");
16
17          btnJava = new JButton("Look & Feel de Java");
18          btnHost = new JButton("Look & Feel del Host");
19          txtInfo=new JTextField("Java",SwingConstants.CENTER);
20          txtInfo.setEnabled(false);
21          pnlPrincipal = new JPanel();
22          pnlPrincipal.add(btnJava);
23          pnlPrincipal.add(new JLabel("Look & Feel Activo"));
24          pnlPrincipal.add(txtInfo);
25          pnlPrincipal.add(btnHost);
26          btnJava.addActionListener(this);
27          btnHost.addActionListener(this);
28          addWindowListener(this);
29          setContentPane(pnlPrincipal);
30
31          pack();
32          setVisible(true);
33    }
34
35    public static void main(String args[])
36    {
37          CambiarLook cl = new CambiarLook();
38    }
39
40    public void actionPerformed(ActionEvent  e)
41    {
42          try
43          {
44                if(e.getSource()==btnJava)
45                {
46
      UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelCl
assName());
47                      txtInfo.setText("Java");
48                }
49                else if(e.getSource()==btnHost)
50                {
51
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName
());
52                      txtInfo.setText("Host");
53                }
```

```
54
55                SwingUtilities.updateComponentTreeUI(this);
56                pack();
57          }
58          catch(Exception ex)
59          {
60                System.out.println("No se cambió el look & feel");
61          }
62    }
63
64    public void windowActivated(WindowEvent e){}
65
66    public void windowClosed(WindowEvent e){}
67
68    public void windowClosing(WindowEvent e)
69    {
70          System.exit(0);
71    }
72
73    public void windowDeactivated(WindowEvent e){}
74
75    public void windowDeiconified(WindowEvent e){}
76
77    public void windowIconified(WindowEvent e){}
78
79    public void windowOpened(WindowEvent e){}
80 }
```

El código relevante está marcado en negrita. En primer lugar en las líneas 46 y 51 establecemos el *look & feel* que convenga en cada caso; a continuación para que los componentes reflejen los cambios debemos invocas al método *updateComponentsTreeUI* para cada container de alto nivel en la jerarquía (*JApplet*, *JDialog* o *JFrame*) como se ve en la línea 55; finalmente se puede llamar al método pack del contenedor principal para ajustar la GUI al cambio de tamaño de los componentes.

10 EL PARADIGMA MODELO-VISTA-CONTROLADOR

Los creadores de *SmallTalk* (uno de los primeros lenguajes orientados a objetos) crearon el paradigma MVC (Model-View-Controller).

Este paradigma consiste en separar la funcionalidad de la interfaz de usuario. Imaginemos por ejemplo que queremos programar una calculadora con una interfaz gráfica.

Cuando nosotros modelemos la clase calculadora, tendremos métodos como sumar, restar, multiplicar, etcétera. Estamos modelando un objeto del mundo real (lo que generalmente se conoce como dominio de la aplicación); llamaremos a esta clase *Calculadora.* Ahora bien, si queremos que el usuario pueda utilizar nuestra calculadora hemos de ofrecerle una manera de hacerlo, imaginemos una interfaz como la calculadora de Windows, por ejemplo. Esta interfaz, o **vista**, la diseñaremos en otra clase que en nuestro ejemplo llamaremos *VistaCalculadora*. Ahora tenemos la funcionalidad y la vista, pero están inconexas. El intermediario entre ambos es lo que conocemos como **controlador**. El controlador se encargará tanto de tratar los eventos que se produzcan en la vista (es decir se encargará de responder a pulsaciones a botones o de actualizar campos de texto, por ejemplo), como de mantener consistente nuestra clase del dominio con respecto a la modificaciones que produce el usuario sobre ella a través de la interfaz. También se encargará de reflejar en la vista las saldas de nuestro programa. El controlador no aporta funcionalidad, sólo aporta el código necesario para que el objeto del dominio y la vista se comuniquen.

Este paradigma resulta muy eficaz porque independiza la interfaz, generalmente muy volátil, de la función con todos los beneficios de reusabilidad que ello comporta. En efecto, si el futuro quisiéramos diseñar una interfaz más elaborada sólo habría que modificar la clase que implementa la vista que la clase dominio deba retocarse. Además el hecho de que la vista no tenga que preocuparse de la funcionalidad hace que la elaboración de interfaces se simplifique y sea más clara.

Con esta división de responsabilidades las clases que componen el dominio nunca deberían preocuparse de tareas como el mantenimiento de pantallas porque éstas son funciones específicas de una aplicación concreta y no al revés. Un objeto del mundo real será utilizado en muchos otros casos y no podemos estar incluyendo dentro de él casos particulares de una funcionalidad de gestión de la interfaz que nos interese sólo en un momento determinado.

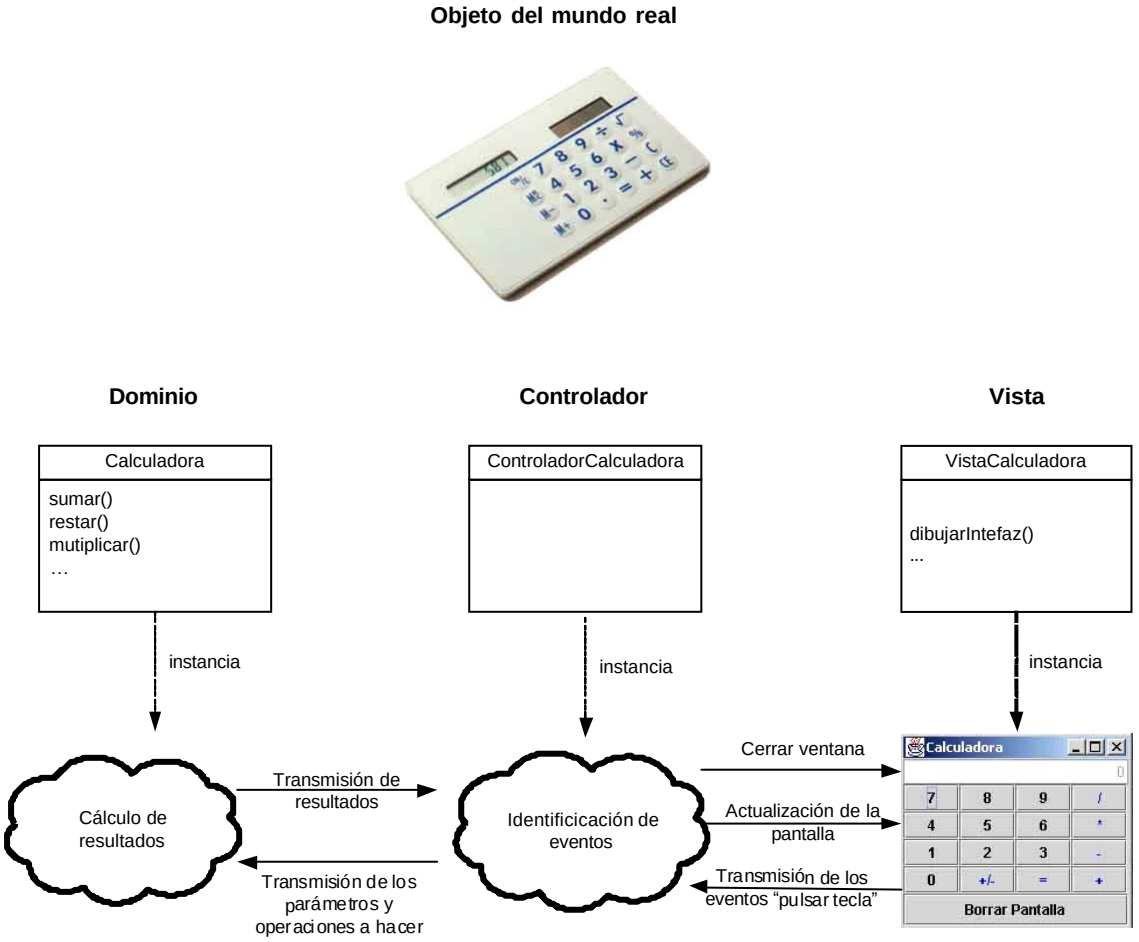


Figura 26
Paradigma M-V-C

10.1Ejemplo comentado

A modo de ejemplo vamos a implementar la anterior calculadora. Empecemos por la clase *Calculadora* que como hemos dicho constituye el dominio de esta aplicación:

```
01    public class Calculadora
02    {
03          private int proximaOperacion; //0=+ 1=- 2=* 3=/ -1=indef.
04          private int a; // primer parámetro
05          private int b; // segundo parámetro
06
07          public Calculadora()
08          {
09                proximaOperacion=-1;
10                a=0;
11                b=0;
12          }
13
14          public int sumar()
15          {
16                return a+b;
17          }
18
```

```
19          public int restar()
20          {
21                return a-b;
22          }
23
24          public int multiplicar()
25          {
26                return a*b;
27          }
28
29          public int dividir() throws ArithmeticException
30          {
31                return a/b;
32          }
33
34          public void setProximaOperacion(int  op)
35          {
36                proximaOperacion = op;
37          }
38
39          public int getA()
40          {
41                return a;
42          }
43
44          public void setA(int a)
45          {
46                this.a=a;
47          }
48
49          public int getB()
50          {
51                return b;
52          }
53
54          public void setB(int b)
55          {
56                this.b=b;
57          }
58
59          public int calcular()
60          {
61                switch(proximaOperacion)
62                {
63                      case 0: return sumar();
64                      case 1: return restar();
65                      case 2: return multiplicar();
66                      case 3: return dividir();
67                      default: return 0;
68                }
69          }
70
71          public String toString()
72          {
73                String str = "a="+a+", b="+b+", proxima
                                        operacion="+proximaOperacion;
74                return str;
75          }
76    }
```

Vemos que esta clase no tiene nada de especial que no hayamos visto hasta ahora. Tiene tres atributos, dos de ellos son los parámetros de la operación y el otro indica qué operación se realizará cuando se llame al método calcular. Pasemos ahora a ver el código de la vista que construirá una GUI como la de la figura 25:

```
001   import javax.swing.*;
002   import java.awt.*;
003
004   public class VistaCalculadora extends JFrame
005   {
006         private JButton btnMas;
007         private JButton btnMenos;
008         private JButton btnPor;
009         private JButton btnDividir;
010         private JButton btnSigno;
011         private JButton btnIgual;
012         private JButton[]btnDigitos;
013         private JButton btnBorrar;
014         private JPanel pnlBotonera;
015         private JTextField txtPantalla;
016         private JPanel pnlPrincipal;
017
018         public VistaCalculadora(ControladorCalculadora  cc)
019         {
020               super("Calculadora");
021
022               txtPantalla = new JTextField("0");
023               txtPantalla.setEnabled(false);
024
      txtPantalla.setHorizontalAlignment(SwingConstants.RIGHT);
025
026               btnMas = new JButton("+");
027               btnMas.setName("+");
028               btnMas.addActionListener(cc);
029               btnMenos = new JButton("-");
030               btnMenos.setName("-");
031               btnMenos.addActionListener(cc);
032               btnPor = new JButton("*");
033               btnPor.setName("*");
034               btnPor.addActionListener(cc);
035               btnDividir = new JButton("/");
036               btnDividir.setName("/");
037               btnDividir.addActionListener(cc);
038
039               btnMas.setForeground(Color.blue);
040               btnMenos.setForeground(Color.blue);
041               btnPor.setForeground(Color.blue);
042               btnDividir.setForeground(Color.blue);
043
044               btnDigitos = new JButton[10];
045
046               for(int i=0;i<10;i++)
047               {
048                     Integer integer = new Integer(i);
049                     btnDigitos[i] = new
JButton(integer.toString());
050                     btnDigitos[i].setName("Digito");
051                     btnDigitos[i].addActionListener(cc);
052               }
053
054               btnSigno = new JButton("+/-");
```

```
055                btnSigno.setName("+/-");
056                btnSigno.addActionListener(cc);
057                btnIgual = new JButton("=");
058                btnIgual.setName("=");
059                btnIgual.addActionListener(cc);
060                btnSigno.setForeground(Color.blue);
061                btnIgual.setForeground(Color.blue);
062
063                pnlBotonera = new JPanel(new GridLayout(4,4));
064                pnlBotonera.add(btnDigitos[7]);
065                pnlBotonera.add(btnDigitos[8]);
066                pnlBotonera.add(btnDigitos[9]);
067                pnlBotonera.add(btnDividir);
068                pnlBotonera.add(btnDigitos[4]);
069                pnlBotonera.add(btnDigitos[5]);
070                pnlBotonera.add(btnDigitos[6]);
071                pnlBotonera.add(btnPor);
072                pnlBotonera.add(btnDigitos[1]);
073                pnlBotonera.add(btnDigitos[2]);
074                pnlBotonera.add(btnDigitos[3]);
075                pnlBotonera.add(btnMenos);
076                pnlBotonera.add(btnDigitos[0]);
077                pnlBotonera.add(btnSigno);
078                pnlBotonera.add(btnIgual);
079                pnlBotonera.add(btnMas);
080
081                btnBorrar = new JButton("Borrar Pantalla");
082                btnBorrar.setName("Borrar");
083                btnBorrar.addActionListener(cc);
084
085                pnlPrincipal = new JPanel(new BorderLayout());
086                pnlPrincipal.add(txtPantalla,"North");
087                pnlPrincipal.add(pnlBotonera,"Center");
088                pnlPrincipal.add(btnBorrar,"South");
089                setContentPane(pnlPrincipal);
090
091                addWindowListener(cc);
092                setSize(200,165);
093                setVisible(true);
094          }
095
096          public JTextField getPantalla()
097          {
098                return txtPantalla;
099          }
100    }
```

En esta clase ya podemos ver cosas más interesantes:

vemos que la constructora, línea 18, recibe una referencia a una instancia de *ControladorCalculadora*. Esto es necesario porque necesitaremos esta referencia para añadir los listeners correspondientes hacia dicha instancia;
a parte de crear todos los componentes de la manera habitual vemos que a muchos de ellos les ponemos un nombre con el método *setName(String nombre)* esto nos servirá, como ya veremos más adelante para poder identificarlos en el controlador;
vemos que añadimos listeners tanto del tipo *actionListener* como *windowListener* (línea 91) al mismo objeto controlador. Esto es posible si dicho

objeto implementa las interfaces necesarias. Recordad que aunque una clase sólo puede heredar de otra, puede implementar tantas interfaces como sea necesario; todos aquellos componentes que pueden generar eventos si no se les pone un listener adecuado dichos eventos no serán tratados por nadie.

Veamos finalmente el código de *ControladorCalculador*:

```
01    import java.awt.event.*;
02    import javax.swing.*;
03
04    public class ControladorCalculadora extends WindowAdapter
implements ActionListener
05    {
06          private Calculadora c;
07          private VistaCalculadora vc;
08          private JTextField txtPantalla;
09          private boolean borrar;
10
11          public ControladorCalculadora()
12          {
13                c = new Calculadora();
14                vc = new VistaCalculadora(this);
15                txtPantalla = vc.getPantalla();
16                borrar=false;
17
18          }
19
20          public void windowClosing(WindowEvent e)
21          {
22                System.exit(0);
23          }
24
25          public void actionPerformed(ActionEvent e)
26          {
27                JButton btn = (JButton) e.getSource();
28
29                if(btn.getName().equals("Digito"))
30                {
31                      if(borrar)
32                      {
33                            txtPantalla.setText("");
34                            borrar=false;
35                      }
36
37                      if(!txtPantalla.getText().equals("0"))
38
      txtPantalla.setText(txtPantalla.getText()+btn.getText());
39                      else
40                            txtPantalla.setText(btn.getText());
41                }
42                else if(btn.getName().equals("Borrar"))
43                {
44                      txtPantalla.setText("0");
45                      c.setProximaOperacion(-1);
46                }
47                else if(btn.getName().equals("+/-"))
48                {
49                      int i =
Integer.parseInt(txtPantalla.getText());
50
51                      if(i>=0)
```

```
52                          i+=-(2*i);
53                    else
54                          i+=-2*i;
55
56
57                    Integer integer = new Integer(i);

58                    txtPantalla.setText(integer.toString());
59              }
60              else if(btn.getName().equals("="))
61              {
62
    c.setB(Integer.parseInt(txtPantalla.getText()));
63
    txtPantalla.setText(Integer.toString(c.calcular()));
64                    borrar=true;
65              }
66              else
67              {
68
    c.setA(Integer.parseInt(txtPantalla.getText()));
69                    borrar=true;
70
71                    if(btn.getName().equals("+"))
72                          c.setProximaOperacion(0);
73                    else if(btn.getName().equals("-"))
74                          c.setProximaOperacion(1);
75                    else if(btn.getName().equals("*"))
76                          c.setProximaOperacion(2);
77                    else if(btn.getName().equals("/"))
78                          c.setProximaOperacion(3);

79              }
80        }
81    }
82
```

Observemos en detalle el código:

nuestra clase implementa los intefaces necesario para gestionar los eventos producidos por los botones y por la ventana. El primero de forma directa implementando la interfaz *actionListener* y la segunda gracias a que nuestro controlador hereda de la clase *WindowAdapter* que a su vez implementa *WindowListener* (línea 4);

nuestro controlador tal como hemos discutido previamente es el intermediario entre la vista y la clase de dominio y por tanto necesitará referencias a estos dos objetos. Hemos optado (líneas 13 y 14) porque sea el propio controlador quien los cree, pero otra solución habría sido por ejemplo que los recibiera como parámetros y fuera otra clase quien se encargara de crearlos (aquí entraríamos en cuestiones de diseño de programas con orientación a objetos y les quitaríamos el trabajo a los profes ingeniería del software, cosa que no pretende este manual);

a continuación redefinimos de la manera habitual el método *windowClosing()* para que la aplicación se termine cuando clickemos sobre la X de la ventana;

el siguiente método es el que se encarga de recibir y gestionar los eventos de pulsación de botón. Dejando al margen la funcionalidad concreta (que no viene al caso), es interesante observar algunos aspectos:

- para identificar qué botón generó el evento utilizamos el método *getName* que devuelve el *String* que le asociamos con el *setName()* en la construcció de la interfaz;
- para poder utilizar los métodos del componente que lo generó primero tenemos que obtenerlo y ello puede hacerse con el método *getSource()* del objeto evento correspondiente. Fijaos que *getSource()* devuelve un *Object* por lo tanto para usar los métodos del componente debermos hacer un casting (línea 27);

Finalmente para poder ejecutar nuestra aplicación deberemos teclear el código siguiente en cualquiera de las clases anteriores, en *Calculadora* por ejemplo:

```
public static void main(String args[])
{
      ControladorCalculadora  cc = new ControladorCalculadora();
}
```

tecleamos a continuación:

```
> javac Calculadora.java
> java Calculadora
```

y aparecerá nuestra supercalculadora totalmente funcional.

Ejercicio 17

Diseñad una interfaz para el banco. Partid del código del ejercicio 13 y no modifiquéis el dominio.

11 APPLETS

Vamos a hablar en este capítulo de un tipo especial de aplicación Java llamada comúnmente *applet*. Patrick Naughton define un *applet* como "una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de un documento web". El propio *web browser* es el encargado de ejecutar el programa, que llega en forma de *bytecode*.

El despegue fulminante de Java se produjo seguramente gracias a los *applets*. Estos prometían dinamizar Internet y aumentar espectacularmente el grado de interactividad de los documentos *web*. Unos pocos años después la realidad es que los browsers modernos soportan HTML dinámico y potentes lenguajes de *scripting* que permiten hacer la mayoría de cosas que hacían los primeros *applets*, y con un rendimiento mejor en la mayor parte de los casos.

Esto no quiere decir que los *applets* hayan dejado de usarse. En Internet su utilización es reducida, y las páginas profesionales con gran número de accesos diarios evitan la inclusión de *applets*. En cambio su uso es cada vez más frecuente en *intranets* o redes corporativas donde el ancho de banda no es un problema. Aprovechando que los *applets* están escritos en un lenguaje de programación de propósito general, plenamente orientado a objetos, podemos construir aplicaciones distribuidas mucho más complejas que lo que permitiría cualquier combinación de HTML y *scripting*. El código de estas aplicaciones es tan fácil de distribuir cómo cualquier documento *web*, es multiplataforma y cualquier ordenador cliente puede ejecutarlas con sólo instalar un *web browser*.

Pese a todo, en este capítulo se explicará cómo crear *applets* con un funcionamiento básico. No es posible explotar al máximo las posibilidades de los *applets* sin usar las potentes capacidades de *red* y *multiflujo* de Java, que van más allá del alcance de este manual (que se explican concienzudamente en el curso de *Java Avanzado* que también imparte Jedi).

Un *applet* es siempre una aplicación gráfica: la interacción básica con el usuario se realiza a través de componentes de GUI clásicos y mediante reacción a movimientos y pulsaciones de ratón u otros eventos similares.

11.1 El *applet* como caso particular de aplicación Swing

Para implementar un *applet* hay que crear una subclase de la clase *javax.applet.JApplet*. Esta clase puede utilizar cualquier otra clase bien sea del J2SDK o hecha por nosotros mismos.

```
java.lang.Object
  |
  +--java.awt.Component
       |
       +--java.awt.Container
            |
            +--java.awt.Panel
                 |
                 +--java.applet.Applet
                      |
                      +--javax.swing.JApplet
```

Como podemos ver un *JApplet* es un Panel, con lo que le podemos añadir componentes *Swing* controlados por un layout manager de la forma habitual, dibujar y mostrar imágenes en su superficie, controlar eventos, etc. De hecho un *applet* es fundamentalmente una aplicación *Swing*, con algunas diferencias:

el *applet* aparece en una ventana del navegador como parte de un documento HTML. Por tanto no requiere la creación de una nueva ventana para mostrarse en ella (aunque si quiere, opcionalmente, puede crear ventanas).;
en algunos navegadores los componentes *Swing* añadidos al *applet* no se muestran hasta que se ha llamado al método validate();
el espacio ocupado por el *applet* se fija en el código del documento HTML en que se incluye, y no en el código del propio *applet*. El *applet* dispone de métodos para consultar su tamaño;
un *applet* no dispone de método *main(...)*. Cuando se carga la página que contiene el *applet*, la Máquina Virtual (que incorpora el propio navegador) crea una instancia de la subclase de *JApplet* que hemos escrito. Después invoca el método *init()* sobre este objeto y acto seguido el método *start()*. Comentaremos enseguida estos métodos.

11.2 El ciclo de vida de un applet

La clase Applet tiene algunos métodos que el *web browser* llama cuando ocurren eventos relevantes de cara al ciclo de vida del *applet*. Applet no hace nada en ninguno de estos métodos, que obviamente es posible sobrescribir para definir el comportamiento del *applet*:

init(): es llamado la primera vez que se carga el documento HTML que hace referencia al *applet*. También en el caso que la página se cargue de nuevo con alguna opción de *reload*. Se usa para hacer inicializaciones como construcción de la jerarquía de componentes que forman la GUI, lectura de parámetros, carga de imágenes, etc. En las subclases de *JApplet* no se suelen escribir constructores. Todo el código que habitualmente pondríamos en un constructor hay que ponerlo en este método porque no existen garantías que el entorno de ejecución sea plenamente funcional hasta que se llama a *init()*;
start(): se llama cada vez que el usuario visita de nuevo la página que contiene el *applet*. En concreto, siempre que se ejecuta *init()* se ejecuta *start()* inmediatamente después. También cuando se vuelve a una página ya visitada o se maximiza una

página que estaba minimizada. Sirve para arrancar la ejecución del *applet*, si este tiene que realizar alguna tarea no ligada estrictamente a algún evento;
stop(). Llamado cuando se abandona la página que contiene el *applet* para visitar otra, o bien cuando se minimiza dicha página. Se usa para suspender cualquier tarea que el *applet* esté realizando y que no tenga sentido continuar mientras el usuario no lo está viendo. Los *applets* que sobreescriben start() suelen sobreescribir también stop() para que el *applet* se ejecute sólo cuando está visible.
destroy(). Llamado cuando se cierra el navegador. Siempre previamente se ha llamado a stop(). Se usa para liberar recursos adicionales que el *applet* pudiera tener, no liberados por stop().

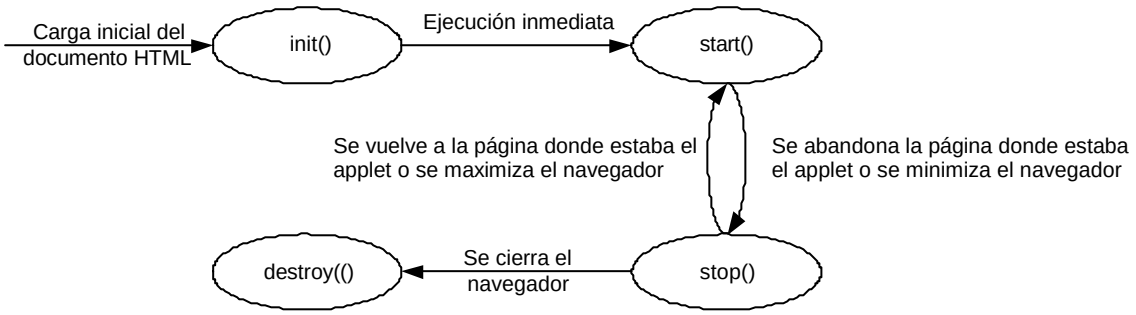


Figura 27
Ciclo de vida de un *applet*

11.3 Diferencias entre *Applet* y *JApplet*

La clase *Applet* es la clase que permite construir estas miniaplicaciones de *AWT* y que por tanto sólo permite utilizar objetos *AWT*; las clase *JApplet* es la versión *Swing* y la que nos permitirá utilizar todas las clases de estas bibliotecas que hemos estudiado en el capítulo precedente. Existen algunas diferencias entre las dos clases que ahora mencionaremos, pero la más obvia es que actualmente no hay ningún navegador que soporte directamente *applets* de *Swing* al contrario que los análogos *AWT*. Para poder utilizar *applets Swing* hay que utilizar el *Java Plug-in* y tenerlo correctamente configurado con nuestros navegadores. El *Java Plug-in* es un componente de la *JRE* de *Sun* o puede ser descargado por separado.

JApplet aporta dos funcionalidades importantes a parte de las que hereda de *Applet.* En primer lugar, los *applets* de *Swing* dan soporte para tecnologías de accesibilidad[4] y en segundo lugar un *JApplet* es un contenedor de alto nivel en la jerarquía y por tanto tiene un *root pane* lo que implica que se pueden incluir menús (esto no era posible con los *applets* de *AWT*). Además por el mismo motivo también tiene un *content pane* lo que implica las siguientes diferencias:

[4] Tecnologías de accesibilidad son todas aquellas ayudas que proporciona un entorno para ayudar a la interacción de personas que tienen disminuidas sus capacidades sensoriales, físicas o mentales de forma temporal o permanente (un ejemplo puede ser soporte de voz para sordos o uso de fuentes grandes para gente con visión reducida).

los componentes deben añadirse al *content pane* y no directamente al *applet;*
debes establecer el *layout manager* deseado sobre el *content pane* y no directamente sobre el *applet*;
por defecto, el *layout manager* del *content pane* es un *BorderLayout* a diferencia que el del *Applet* que es un *FlowLayout*;
no es aconsejable (en algunos navegadores no se visualizará) "dibujar" directamente sobre el *applet*.

11.4 Ejemplo

Para hacer un *applet* basta, como hemos dicho, con crear una subclase de *JApplet.* Como mínimo habrá que sobrescribir el método *init().* En algunos casos hay que sobrescribir también otros métodos relacionados con el ciclo de vida del *applet,* y a menudo métodos relacionados con el dibujo en pantalla.

Vamos a ver un ejemplo sencillo de *applet* que sobrescribe todos los métodos relacionados con el ciclo de vida, además del método *paint(...)* que hereda de las clases *Swing*. Simplemente muestra un *String* en pantalla indicando los eventos relacionados con el ciclo de vida que se van sucediendo:

```
import java.awt.*;
import javax.swing.*;

public class PrimerApplet extends JApplet
{
      private String buffer; private
      JLabel lblMensaje; private
      Container cntPrincipal;

      public void init()
      {
            buffer = new String();
            lblMensaje = new JLabel();
            cntPrincipal = getContentPane();
            cntPrincipal.add(lblMensaje);
            establecerMensaje("Inicializando...");
      }

      public void start()
      {
            establecerMensaje("Empezando...");
      }

      public void stop()
      {
            establecerMensaje("Parando...");
      }

      public void destroy()
      {
            establecerMensaje("Preparandose  para descargar de
memoria...");
      }

      private void establecerMensaje(String  nuevoMensaje)
```

```
      {
            System.out.println(nuevoMensaje);
            buffer += nuevoMensaje;
            lblMensaje.setText(buffer);
      }


}
```

Tal como ya se ha dicho, un applet sólo puede verse a través de la página web que lo contiene. Ahora en el siguiente apartado veremos cómo se insertan en una web.

11.5 El tag <APPLET> de HTML

Para usar un *applet* a través del web es necesario tener un fichero HTML que haga referencia al mismo. Esa referencia se lleva a cabo con la marca <APPLET>. De hecho la única forma de ejecutar un *applet* es visualizar la página HTML donde se le hace referencia con un navegador o con algún otro programa capaz de interpretar la marca <APPLET> de HTML, como el *appletviewer* que viene con el J2SDK.

La forma más simple de la marca <APPLET> (si falta alguno de los atributos no funcionará) es la siguiente:

```
<APPLET CODE = AppletSubclase WIDTH = anchura HEIGHT = altura>
</APPLET>
```

así pues para poder utilizar nuestro applet deberíamos crear un archivo HTML como el que sigue:

```
<HTML>
<HEAD>
      <TITLE>¡Probando APPLETS!<TITLE>
<HEAD>
<BODY>
      <APPLET CODE=PrimerApplet.class WIDTH=200 HEIGHT=200>
      </APPLET>
</BODY>
</HTML>
```

ahora sólo resta visualizar nuestro *applet* con el *appletviewer*. Si queremos visualizarlo con el navegador, consecuencia del no soporte nativo en los mismos, para visualizarlo además de tener instalado el *Java Plug-in* deberemos parsear el HTML con el *HTML Conversor*. Esto se explica en el último apartado de este capítulo dedicado a los applets.

11.5.1 Atributos del tag <APPLET>

La forma general del tag <*APPLET*> es la siguiente:

```
<APPLET
    [CODEBASE = codebaseURL]
    CODE = appletFile
    [ALT = alternateText]
    [NAME = appletInstanceName]
    WIDTH = pixels
    HEIGHT = pixels
    [ALIGN = alignment]
    [VSPACE = pixels]
    [HSPACE = pixels]
>
[< PARAM NAME = appletParameter1  VALUE = value >]
[< PARAM NAME = appletParameter2  VALUE = value >]
. . .
[alternateHTML]
</APPLET>
```

`CODEBASE = codebaseURL`
Este atributo opcional especifica el URL básico del *applet*: el directorio que contiene el código del *applet* (los ficheros `.class`). Si no se especifica este atributo se usa el URL del propio documento HTML. Cuando un *applet* trata de usar una clase por primera vez, la busca primero en el host que está ejecutando el *web browser*. Si la clase no está allí –las clases de librería del JDK suelen estar- busca el *bytecode* en el *host* de donde proviene el *applet*, con direcciones relativas a este URL básico.

`CODE = appletFile`
Este atributo obligatorio indica el nombre del fichero que contiene la subclase de `Applet` compilada. El nombre es relativo al URL básico (no puede ser absoluto).

`ALT = alternateText`
Atributo opcional donde se puede escribir cualquier texto que el *browser* mostrará si es capaz de interpretar la marca `<APPLET>` pero por alguna razón no puede ejecutar *applets* Java.

`NAME = appletInstanceName`
También opcional. Especifica un nombre para la instancia de *applet*, que puede ser usado por otros *applets* en la misma página para localizar el *applet* y comunicarse con él.

`WIDTH = pixels`
`HEIGHT = pixels`
Estos atributos obligatorios dan la anchura y altura iniciales (en *pixels*) del área que el *applet* tiene disponible para mostrarse, sin incluir posibles ventanas y diálogos que el *applet* puede mostrar.

`ALIGN = alignment`
Opcional. Especifica el alineamiento del *applet*. Valores posibles (los mismos que para la marca `<IMG>` y con el mismo efecto): `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, `absbottom`.

`VSPACE = pixels`
`HSPACE = pixels`
Opcionales. Especifican el número de *pixels* por encima y por debajo (`VSPACE`) y a cada lado (`HSPACE`) del *applet*. Son tratados de la misma forma que los atributos homólogos de `<IMG>`.

`<PARAM NAME = appletParameter1 VALUE = value>`
Las marcas `<PARAM>` son la única forma de pasar parámetros al *applet* (no existe llamada desde línea de comandos). En próximo apartado se explica como el *applet* puede leer y usar esta información.

`alternateHTML`
Si la página está siendo mostrada por un *browser* que no entiende la marca `<APPLET>`, este texto será mostrado en su lugar y el resto de código entre `<APPLET>` y `</APPLET>` será ignorado. En caso contrario será este texto lo que el *browser* ignorará. Se puede usar esto para dar un mensaje del estilo "Aquí debería haber un applet pero tu browser no es capaz de mostrar applets".

11.5.2 Paso de parámetros a un applet

Algunos *applets* permiten al usuario que los incluye en su documento HTML configurarlo mediante parámetros con el tag `<PARAM>`.

El *applet* puede usar el siguiente método de *JApplet* para leer el valor de estos parámetros:

```
public String getParameter(String name)
```

Aunque los parámetros se recogen en forma de *`String`*, se pueden convertir, obviamente, a cualquier otro tipo que sea necesario, como por ejemplo un *int.*

Veamos un ejemplo de uso de parámetros para hacer el tipo y el tamaño de las fuentes configurables por el usuario:

```
<APPLET CODE="FontParamApplet.class" WIDTH=200 HEIGHT=200>
<PARAM NAME=font VALUE="Helvetica">
<PARAM NAME=size VALUE="24">
</APPLET>
```

```
public class FuenteParamApplet extends JApplet
{
      public void init()
{
            String fontName = getParameter("font");
            int fontSize = Integer.parseInt(getParameter("size"));
            ...
      }
}
```

Es importante que en código del *applet* se prevea un comportamiento por defecto en caso que el usuario no especifique valor para alguno de los parámetros (o especifique un valor incorrecto).

11.6 Restricciones por razones de seguridad

En teoría un *applet* es una aplicación Java como cualquier otra. Pero ejecutar código que llega directamente de la red es un riesgo de seguridad clásico. Por eso los navegadores imponen ciertas restricciones a los applets que son cargados desde la red:

un *applet* no puede ejecutar código que no sea Java;
no puede leer o escribir en ficheros del sistema que lo ejecuta (el cliente de *web*);
no puede establecer conexiones de red excepto con el host de dónde procede;
no puede ejecutar ningún otro programa en el sistema que lo ejecuta;
las ventanas que el *applet* abre tienen un aspecto distinto a las de las aplicaciones normales para evitar que un usuario despistado escriba datos comprometedores en ellas confundiéndolas con algún diálogo del sistema (normalmente llevan un mensaje debajo del estilo: "cuidado, esta ventana la está ejecutando un applet" o "subprograma iniciado").

Cada navegador tiene un objeto *SecurityManager* que implementa la política de seguridad (con ligeras variaciones de un navegador a otro). Cuando el *applet* intenta realizar una operación que tiene prohibida se lanza una *SecurityException*, que como cualquier otra excepción puede ser capturada y tratada por el *applet.*

Todas estas restricciones no se aplican a *applets* cuyo código se encuentra en el sistema de ficheros local (en un directorio accesible mediante el CLASSPATH).

11.7 Algunas capacidades interesantes

Llamando al método de Applet

```
AppletContext getAppletContext()
```

se obtiene un objeto representando el entorno de ejecución del *applet*, el navegdor, con el cual se pueden hacer cosas interesantes. Por ejemplo, se puede forzar que el navegador muestre un documento *web* en alguna de sus ventanas con los métodos de AppletContext

```
public void showDocument(java.net.URL url)
public void showDocument(java.net.URL url, String targetWindow)
```

También es posible comunicarse con otros *applets* referenciados en el mismo documento HTML usando

```
Enumeration getApplets()
Applet getApplet(String name)
```

El primero retorna una enumeración con todos los objetos *Applet* de la página y el segundo el *Applet* con el nombre especificado (recordar que se da nombre a un *applet* mediante el atributo NAME de la marca <APPLET>). Una vez se consigue una referencia al *Applet* es posible invocar cualquier método sobre el objeto para pasarle o pedirle información.

Otra facilidad interesante es la que proporcionan los métodos de Applet

```
Image getImage(URL url)
```

```
Image getImage(URL url, String name)
AudioClip getAudioClip(URL url)
AudioClip getAudioClip(URL url, String name)
```

que permiten cargar ficheros con imágenes (.GIF o .JPEG) o sonidos (.AU) a partir de un URL (que no es más que un nombre para localizar un recurso en Internet). Java es capaz de mostrar estas imágenes y ejecutar estos ficheros de sonido.

Para usar estos métodos puede ser necesario llamar a

```
public URL getDocumentBase()
```

que da la página que contiene el *applet*, o bien a

```
public URL getCodeBase()
```

que retorna URL que contiene el *bytecode* del *applet* mismo.

Se recomienda consultar la API para usar todas estas facilidades.

Ejercicio 11.1

Convertid el ejercicio 9.3 en un *JApplet* y cread un documento HTML para poder visualizarlo con el *appletviewer*.

Ejercicio 11.2

Haced un applet *MuestraTexto* que reciba dos parámetros, *Mensaje* y *Color* (que puede tomar los valores rojo, verde y azul) y que en función de los mismos muestre un texto con los valores correspondientes. Tened en cuenta que el usuario podría obviar el valor de alguno o de todos los parámetros o que podría introducir valores incorrectos. Comprobad que funciona creando un documento HTML adecuado.

11.8 Utilizando el *HTML Converter*

El *HTML Converter* es un parser construido en Java (aunque existan versiones compiladas para distintas plataformas) que dado un documento HTML obtiene otro donde se han añadido las marcas necesarias para poder visualizar los *applets* de *Swing* en los browsers más conocidos. Además, como ya se ha dicho varias veces, deberemos tener instalado correctamente el *Java Plug-in*.

El parser es una aplicación gráfica cuya pantalla principal es la que se muestra en la figura 28.

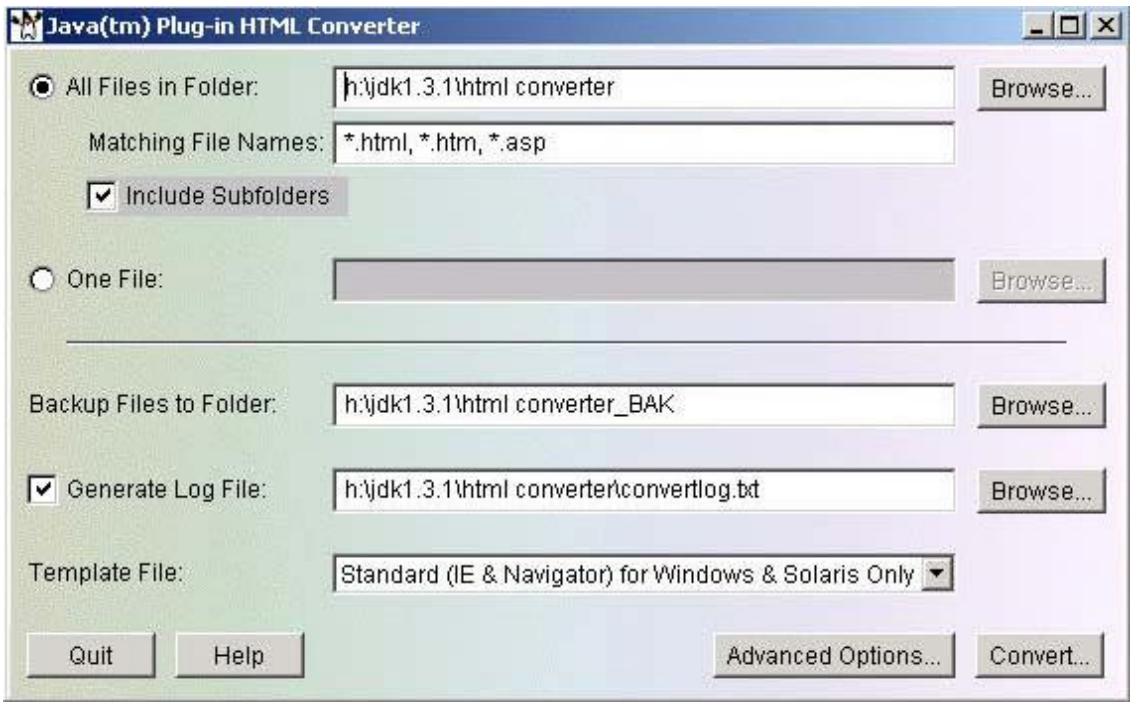


Figura 28
HTML Converter

Su utilización es muy sencilla e intuitiva: sólo hay que seleccionar el directorio de donde están los documentos HTML, donde guardar los ficheros de backup, si queremos generar un archivo de log (y su ruta) y qué tipo de documento HTML queremos generar.

Este es un programa gratuito que puede descargarse de la web de Sun en http://java.sun.com/products/plugin/converter.html.

12 ENTRADA/SALIDA: STREAMS

La entrada salida de Java se basa en el uso de flujos de datos también conocidos como *streams*. Los *streams* son unas secuencias de bytes que viajan desde un origen a un destino a través de una vía de comunicación. Estos flujos puedes ser de lectura y/o escritura y se pueden conectar a distintos fuentes y destinos de datos pudiendo consistir en, por ejemplo, transferencias de memoria a memoria, sistemas de archivos, redes y otras formas de E/S.

La idea principal es que usamos los flujos para leer y escribir datos en los distintos dispositivos a través de sus métodos para poder abstraernos de las características concretas de la vía de comunicación. Como resultado de esto tenemos una serie común de métodos para leer y escribir, métodos que se sobrescriben y aumentan dependiendo del nivel de abstracción del *stream* con el que trabajemos; por ejemplo: podemos tener *streams* que lean bytes directamente y otros que permitan escribir un objeto sin más sin tener que hacer ninguna transformación previa.

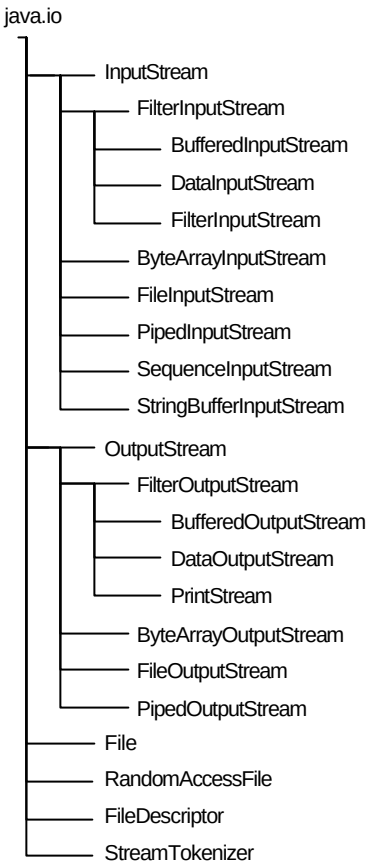


Figura 29
Jerarquía de *java.io*

En la figura de la derecha podemos ver la jerarquía principal del paquete *java.io* donde están definidos los *streams*. Como puede verse se tienen dos *streams* principales: *InputStream* y *OutputStream*. Otras clases de nivel superior son *File*, *FileDescriptor*, *RandomAccessFile* y *StreamTokenizer* Tanto *InputStream* como *OutputStream* tienen un conjunto de subclases complementarias.

Algunos *streams* son **nodos** (*node stream*), que significa que leen o escriben en un sitio específico, como puede ser un archivo de disco o un espacio de memoria. Otros *streams* se llaman **filtros** (*filters*), y sirven para poder leer o escribir datos con un cierto formato (por ejemplo en formato ZIP, o con el formato de los tipos primitivos de Java). Un *stream* de entrada de tipo filtro se crea con una conexión a un *stream* de entrada ya existente. Esto está hecho así para que cuando se intente leer de un objeto filtro de entrada, éste eche mano de caracteres que vengan de otro objeto *stream* de entrada. Este mecanismo de encadenamiento (o apilamiento) de los *streams* permite leer/escribir en cualquier formato de/sobre cualquier tipo de dispositivo.

12.1 InputStream

Los métodos de lectura del *stream* básico de lectura son:

```
public int read()
public int read(byte[] b)
int read(byte[] b, int desplazamiento, int longitud)
```

que permiten el acceso al flujo de datos generador de bytes. El primer método lee un sólo byte y lo devuelve como un entero en el rango 0-255 (si queremos utilizarlo como byte deberemos forzar un casting); si devuelve un –1 quiere decir que hemos alcanzado el fin de la fuente de datos (si estuviera conectado a un fichero, por ejemplo, sería la marca de fin de fichero, el EOF de otros lenguajes de programación). Los otros dos métodos acceden a los datos mediante un vector de bytes: los datos que leamos desde el *stream* se almacenarán en el mismo y el valor de retorno debe interpretarse si es diferente de –1 como el número de datos que hemos podido leer y si es igual a –1 como que se ha alcanzado el fin de la fuente de datos. El segundo método leerá tantos bytes como le sea posible y el tercero intentará leer con un desplazamiento inicial (segundo parámetro) tantos bytes como se le indican (tercer parámetro). Todos estos métodos son bloqueantes, lo que quiere decir que una vez los invoquemos no devolverán el valor a la aplicación hasta que puedan leer algo o hasta que se produzca una excepción porque se cierre el *stream* por ejemplo.

Cuando se ha terminado de usar un *stream*, éste se tiene que cerrar. Esto se hace con el método:

```
public void close()
```

Si se tiene una pila de *streams* (en seguida veremos lo que es), con filtros incluidos, se tiene que cerrar el *stream* de la parte superior de la pila. Esta operación cierra los demás *streams*

A veces nos interesa saber qué número de bytes están preparados en el *stream* para ser leídos. Esto se hace con el método:

```
public int available()
```

que devuelve precisamente eso. Una posterior lectura del *stream* leerá al menos tantos bytes por los indicados en el valor de retorno de una llamada precedente a este método.

Si queremos descartar un número de bytes de nuestro *stream* porque no nos interesen deberemos usar el método

```
public void skip(long numBytes)
```

Finalmente tenemos los métodos

```
public boolean markSupported()
public void mark(int numBytes)
public void reset()
```

que nos permiten implementar operaciones de *push back*. Este tipo de operaciones son las que nos permiten marcar un número de bytes a partir de una posición para más adelante, una vez se haya desplazado el puntero de lectura/escritura localizarla rápidamente. Este tipo de operaciones no están disponibles en todas las fuentes de datos; para detectar si el *stream* actual está conectado a una fuente que permite este tipo de operacioens tenemos el primer método *markSupported()*. El método *mark(...)* marca, valga la redundancia, la posición

actual así como el número de bytes que se podrán leer después de resituar el *stream* a dicha marca y finalmente el método *reset()* resitúa el *stream* a la posición previamente marcada.

12.2 OutputStream

Análogamente al *InputStream* tenemos los métodos de escritura siguientes:

```
public void write(int b)
public void write(byte[] b)
public void write(byte[], int desplazamiento, int longitud)
```

el primero escribe un byte correspondiente al byte de menos peso del parámetro siendo los otros totalmente ignorados. El segundo método escribe b.length bytes del array b y el tercero escribe tantos bytes del array b como se indica en el tercer parámetro desde la posición indicada por el segundo.

Disponemos también del método de cierre del *stream:*

```
public void close()
```

al igual que en el caso anterior si tenemos una pila de *streams* el cierre del que hay en el top de la misma cierra todos.

Algunas veces los *streams* de escritura acumulan los datos de sucesivas escrituras en estructuras de memoria antes de escribirlas realmente por motivos de *buffering* y eficiencia. Si queremos estar seguros de que nuestra escritura se produce en ese momento podemos forzarlo explícitamente con el método:

```
public void flush()
```

De todos modos se garantiza un *flush()* implícito cuando se llama a *close()*.

12.3 Otras clases básicas de *streams*

Como se ha dicho, en el paquete *java.io* hay definidas varias clases de *streams*. Todos los *streams* de entrada son descendientes de *InputStream* y todos los de salida lo son de *OutputStream.* Algunos de las más comunes se describen a continuación.

FileInputStream y *FileOutputStream* son *streams* de tipo nodo que utilizan ficheros del disco como fuente/destino de los bytes. Los constructores de estas clases permiten especificar el camino del fichero al que se conectan. Para construir un *FileInputStream*, el fichero asociado tiene que existir y tiene que poderse leer. Si se construye un *FileOutputStream*, el fichero de salida, si ya existe, se sobrescribirá y en caso de que no exista se creeará. Ejemplo de su utilización podrían ser los siguientes:

```
FileInputStream infile = new FileInputStream("myfile.dat");
FileOutputStream outfile = new FileOutputStream("results.dat");
```

DataInputStream y *DataOutputStream* son *stream* de tipo filtro que permiten leer y escribir tipos primitivos de Java. Existen métodos especiales para cada tipo. Por ejemplo, en *DataInputStream* tenemos:

```
public byte readByte()
public long readLong()
public double readDouble()
```

y en *DataOutputStream*:

```
public void writeByte(byte)
public void writeLong(long)
public void writeDouble(double)
```

Estos *streams* tienen métodos para leer y escribir cadenas de caracteres (String), pero como se comenta en un apartado posterior no permiten trabajar con codificación ASCII ya que recordad que Java no utiliza este códi para representar el juego de caracteres sino que utiliza la extensión Unicode.

En este apartado se han comentado los métodos más relevantes de cada una de las clases mencionadas. Se invita al lector a que investigue en la documentación de la API qué más métodos y funcionalidades aportan las mismas.

12.4 Encadenar *streams*

Como ya se ha comentado anteriormente, Java nos permite encadenar (o apilar según se mire) *streams* de tipo filtro encima de otros de tipo nodo (o de tipo filtro) para lograr un nivel de abstracción más alto y poder leer/escribir estructuras cada vez más complejas. Por ejemplo, podríamos encadenar un *stream* de tipo *DataInputStream* sobre un *FileInputStream* para poder leer directamente tipos primitivos desde un fichero.

Si os fijáis con atención en la documentación de la API veréis que todos los *streams* de tipo filtro reciben en su constructor otro *stream*. Este es el sistema que se usa para encadenar *streams*. Veamos un ejemplo completo autoexplicativo en que se crean unos cuantos tipos primitivos, se muestran por pantalla, se escriben en un fichero, se leen después del mismo y a continuación se vuelven a mostrar por pantalla:

```
import java.io.*;

public class EscribirLeerFichero
{
  public static void main(String args[])
  {
    boolean b=true;
    int i=10;
    double d=5.5;
    String str = "Hola Mundo!";

    System.out.println("Antes de la escritura en el fichero...");
    System.out.println("El booleano b vale: "+b);
    System.out.println("El int i vale: "+i);
    System.out.println("El double d vale: "+d);
    System.out.println("El String str vale: \""+str+"\"\n");

    System.out.println("Haciendo operaciones de esntrada/salida sobre
                                                    los ficheros...");

    try
    {
```

```
      // Creo el stream de nodo conectado
// al fichero para escribir en él
      FileOutputStream fos = new FileOutputStream("datos.dat");
      // Le encadeno el stream de datos para
// poder escribir tipos primitivos
      DataOutputStream dos = new DataOutputStream(fos);

      // Hago las escrituras correspondientes
      dos.writeBoolean(b);
      dos.writeInt(i);
      dos.writeDouble(d);
      dos.writeUTF(str);

      // Fuerzo que se escriba todo con un flush aunque no sería
      // necesario porque la siguiente instrucción es un close
      // y se haría de forma implícita
      dos.flush();

      // Cierro el stream filtro que al ser el top de la pila
      // cierra también el stream nodo
      dos.close();

      // Creo el stream de nodo conectado al fichero para leer de él
      FileInputStream fis = new FileInputStream("datos.dat");
      // Le encadeno el stream de datos para
// poder leer tipos primitivos
      DataInputStream dis = new DataInputStream(fis);

      // Hago las lecturas correspondientes
      b = dis.readBoolean();
      i = dis.readInt();
      d = dis.readDouble();
      str = dis.readUTF();
    }
    catch(IOException e)
    {
      System.out.println("Se ha producido la execpción de
                                        entrada/salida siguiente: "+e);
    }

    System.out.println("Después de la escritura y lectura en el
                                                        fichero...");
    System.out.println("El booleano b vale: "+b);
    System.out.println("El int i vale: "+i);
    System.out.println("El double d vale: "+d);
    System.out.println("El String str vale: \""+str+"\"\n");
  }
}
```

12.5 Lectura/escritura en formato ASCII

Como ya se ha dicho en numerosas ocasiones a lo largo de la presente documentación, Java utiliza Unicode para representar caracteres utilizando dos bytes por carácter en vez de uno como en ASCII. Esto no supone ningún problema mientras que los ficheros generados se vayan a utilizar en programas Java. El problema surge cuando queremos generar (o leer) ficheros de datos que podrán ser la entrada de programas escritos en otros lenguajes y que utilicen codificación ASCII.

Para solucionar este problema se utilizan los lectores y escritores que heredan de *Reader* y *Writer*. Como en los *streams*, en el paquete *java.io* se puede encontrar una gran variedad de estos lectores y escritores.

Las versiones más importantes de lectores y escritores son *InputStreamReader* y *OutputStreamWriter*. Estas clases se usan como interfaz entre *streams* de bytes y lectores y escritores de caracteres.

Cuando se construye un *InputStreamReader* o un *OutputStreamWriter*, se definen una serie de reglas de conversión entre Unicode de 16 bits y las representaciones específicas en otras plataformas.

12.6 Canales de entrada y salida estándar y de error

Todas las aplicaciones tienen definidos tres canales:

el de **entrada** estándar: canal de donde por defecto se leen los datos de entrada y que está en principio conectado al teclado;
el de **salida** estándar: canal donde se vuelcan los resultados por defectos y que está conectado al monitor;
y el de **error** que es por el cual se vuelcan los posibles mensajes de error.

En java estos canales están implementados como *streams* y pueden accederse a través de la clase *System* en donde están definidos como atributos. En concreto:

Tabla 8. Canales Estándar

Canal	Accesible desde	Tipo de stream
Entrada	System.in	InputStream
Salida	System.out	PrintStream
Error	System.err	PrintStream

Podemos ver que en concreto el canal de entrada es un *InputStream* y que tanto el de salida como el de error son *streams* de tipo *PrintStream* que son subclases (no directas) de *DataOutputStream.*

Como decíamos estos canales están conectados por defecto a los que se conoce como consola (teclado más monitor), pero esto puede modificarse con los métodos *setIn(...), setOut(...)* y *setErr(...)* que reciben como parámetros *streams* del tipo adecuado.

12.7 Ficheros

Hablaremos aquí de algunas utilidades de Java para trabajar con el sistema de ficheros local.

12.7.1 Creación de un objeto fichero

La clase *File* proporciona algunas utilidades para trabajar con ficheros y obtener información básica sobre ellos.

```
File miFichero;
miFichero = new File("miFichero");
```

```
miFichero = new File(File.separator, "miFichero");
// Es mucho más útil si el directorio o nombre del fichero es una
// variable
// File.separator da una representación en forma de string del
// separador de nombres de directorio (p.e. "/"). Puede ser
// distinto según en qué plataforma se ejecute esto
File miDirectorio = new File(File.separator);
miDirectorio = new File(myDir, "miFichero");
```

Si se utiliza un sólo fichero en una aplicación, se usa el primer constructor. Si por el contrario, se utilizan varios ficheros de un mismo directorio común, es más fácil utilizar el segundo o tercer constructor.

La clase *File* define métodos, que son independientes de la plataforma en la que se trabaja, para manejar un fichero que pertenece a un sistema de ficheros no propietario. De todas formas, no permite el acceso a los contenidos del fichero (que debe realizarse mediante *streams*).

Se puede utilizar un objeto File como argumento del constructor de los objetos *FileInputStream* y *FileOutputStream* en vez de un *String*. Esto permite independizarse de las convenciones del sistema de ficheros local y generalmente es recomendable.

12.7.2 Comprobaciones y utilidades sobre los ficheros

Una vez creado el objeto File, se pueden utilizar los siguientes métodos para reunir información sobre el fichero:

Los nombres de fichero:

```
public String getName()
public String getPath()
public String getAbsolutePath()
public String getParent()
public boolean renameTo(File newName)
```

Comprobaciones sobre ficheros:

```
public boolean exists()
public boolean canWrite()
public boolean canRead()
public boolean isFile()
public boolean isDirectory()
public boolean isAbsolute();
```

Información general sobre ficheros y utilidades:

```
public long lastModified()
public long length()
public boolean delete()
```

Utilidades de directorios:

```
public boolean mkdir()
public String[] list()
```

12.8 La interfaz *Serializable*

Los objetos que son instancias de clases que implementan la interfaz *java.io.Serializable* tienen la posibilidad de ser almacenados o enviados por una línea de comunicaciones como si fueran un tipo primitivo más, es decir tendremos métodos que recibirán como parámetro un objeto serializado y lo escribirán, por ejemplo, en un fichero sin que el usuario tenga que preocuparse de cómo se hace. Guardar un objeto en cualquier tipo de memoria permanente recibe el nombre de **persistencia**.

La interfaz *Serializable* no tiene métodos y sólo sirve como marca, es decir, la clase que implementa esta interfaz será considerada para serialización. Los objetos cuyas clases no implementen esta interfaz no pueden guardar o recuperar sus estados.

En un objeto serializado se guardan sólo los atributos del objeto; los métodos no forman parte del *stream* que se serializa. Cuando un atributo del objeto que se serializa es también un objeto, todos los atributos de este último se serializan también. El árbol o estructura de un objeto con sus atributos, incluyendo los subobjetos que contiene, constituye la representación interna de un objeto serializado.

Algunas clases no son serializables porque los datos que representan están en constante cambio; por ejemplo los *streams*. Si un objeto serializable contiene un puntero hacia un objeto no serializable, toda la operación de serialización falla y se lanza la excepción *NotSerializableException*.

Si la representación interna de un objeto serializado contiene un puntero a un objeto no serializable, el primer objeto, todavía se puede serializar si el objeto no serializable se marca con la palabra reservada *transient.* Por ejemplo:

```
public class MiClaseSerializable implements Serializable
{
      public transient InputStream entradaDatos;
      public String nombre;
      ...
      public MiClaseSerializable(...)
      {
            ...
      }
      ...
}
```

Los modificadores de visibilidad no afectan los distintos datos que se serializan. Los datos se escriben en el *stream* en formato de byte y con strings representados como caracteres UTF[5]. La palabra reservada *transient* aplicada a un dato, evita que ese dato se serialice.

12.8.1 ObjectInputStream y ObjectOutputStream

ObjectInputStream y *ObjectOutputStream* son los dos *streams* de tipo filtro que nos permiten leer y escribir objectos serializables de forma directa y sencilla a través de los métodos

[5] UTF: Universal character set Transformation Format

```
        public Object readObject()
        public void writeObject(Object obj)
```

Como ejemplo vamos a ver un programa que mira la hora actual y la almacena en un fichero para después poder utilizarla:

```
import java.io.*;
import java.util.Date;

public class AlmacenarFecha
{
    public static void main(String args[])
    {
        Date ahora = new Date();
        System.out.println("Voya a almacenar la fecha: "+ahora);

        try
        {
            FileOutputStream fos = new FileOutputStream("fechas.dat");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ahora);
            oos.close();
        }
        catch(IOException e)
        {
            System.err.println("Se ha producido una excepción: "+e);
        }


        Date entonces;

        try
        {
            FileInputStream fis = new FileInputStream("fechas.dat");
            ObjectInputStream ois = new ObjectInputStream(fis);
            entonces = (Date) ois.readObject();
            ois.close();
            System.out.println("He recuperado la fecha: "+entonces);
        }
        catch(Exception e)
        {
            System.err.println("Se ha producido una excepción: "+e);
        }
    }
}
```

13 ÍNDICE DE FIGURAS

14 ÍNDICE DE TABLAS

15 LINKS DE INTERÉS

www.sun.com: web de Sun, desarrolladores de Java. java.sun.com: web principal de Java http://java.sun.com/j2se/1.4/docs/api/index.html: documentación de la API http://java.sun.com/docs/books/tutorial/ : Java Tutorial
http://java.sun.com/products/plugin/converter.html: página de download del HTML Converter

16 BIBLIOGRAFÍA

Jamie Jaworski: *Java Guía de desarrollo*. Prentice Hall mayo, 1997, España
Curso de Java. Jedi, junio 2001, Barcelona
Curso de Java Avanzado. Jedi, febrero 2001, Barcelona
web de Java de Sun: *http://java.sun.com/*
The Java Tutorial (http://java.sun.com/docs/books/tutorial/)