Java2

incluye Swing, Threads, programación en red, JavaBeans, JDBC y JSP / Servlets

Autor: Jorge Sánchez (www.jorgesanchez.net) año 2004

Basado en el lenguaje Java definido por Sun (http://java.sun.com)

índice

introducción

historia de Java

los antecedentes de Java

Java es un lenguaje de programación creado para satisfacer una necesidad de la época (así aparecen todos los lenguajes) planteada por nuevos requerimientos hacia los lenguajes existentes.

Antes de la aparición de Java, existían otros importantes lenguajes (muchos se utilizan todavía). Entre ellos el lenguaje C era probablemente el más popular debido a su versatilidad; contiene posibilidades semejantes a programar en ensamblador, pero con las comodidades de los lenguajes de alto nivel.

Uno de los principales problemas del lenguaje C (como el de otros muchos lenguajes) era que cuando la aplicación crecía, el código era muy difícil de manejar. Las técnicas de programación estructurada y programación modular, paliaban algo el problema. Pero fue la **programación orientada a objetos (POO** u **OOP)** la que mejoró notablemente el situación.

La POO permite fabricar programas de forma más parecida al pensamiento humano. de hecho simplifica el problema dividiéndolo en objetos y permitiendo centrarse en cada objeto, para de esa forma eliminar la complejidad. Cada objeto se programa de forma autónoma y esa es la principal virtud.

Al aparecer la programación orientada a objetos (en los ochenta), aparecieron varios lenguajes orientados a objetos y también se realizaron versiones orientadas a objetos (o semi—orientadas a objetos) de lenguajes clásicos.

Una de las más famosas fue el lenguaje orientado a objetos creado a partir del C tradicional. Se le llamó C++ indicando con esa simbología que era un incremento del lenguaje C (en el lenguaje C, como en Java, los símbolos ++ significan incrementar). Las ventajas que añadió C++ al C fueron:

- Añadir soporte para objetos (POO)
- Los creadores de compiladores crearon librerías de clases de objetos (como **MFC**[1] por ejemplo) que facilitaban el uso de código ya creado para las nuevas aplicaciones.
- Incluía todo lo bueno del C.

C++ pasó a ser el lenguaje de programación más popular a principios de los 90 (sigue siendo un lenguaje muy utilizado).

Otras adaptaciones famosas fueron:

- El paso de Pascal a Turbo Pascal y posteriormente a Delphi.
- El paso de Basic a QuickBasic y después a Visual Basic.

Pero el crecimiento vertiginoso de Internet iba a propiciar un profundo cambio.

[1] ***Microsoft Foundation Classes***, librería creada por Microsoft para facilitar la creación de programas para el sistema Windows.

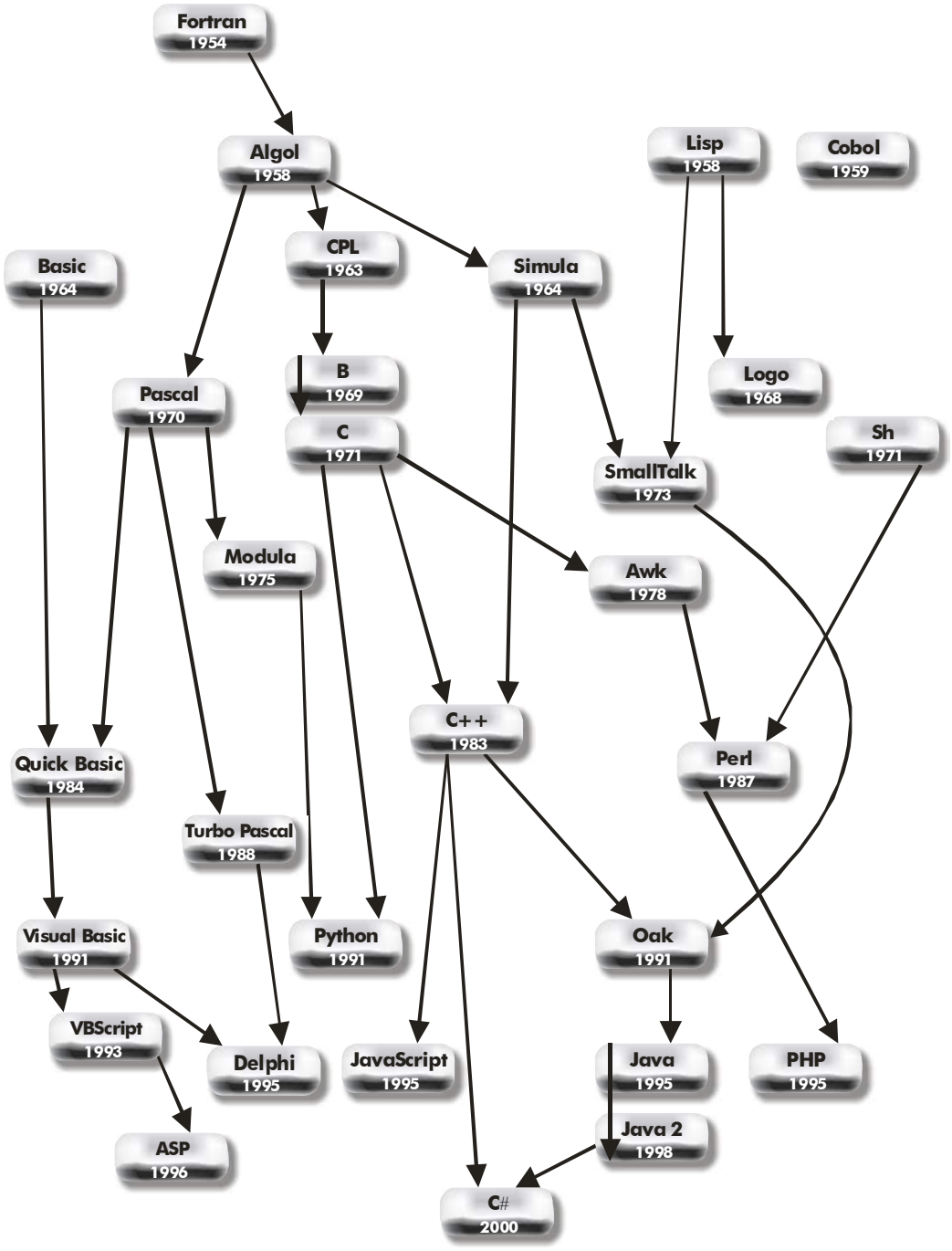


Ilustración 1, ***Evolución de algunos lenguajes de programación***

la llegada de Java

En 1991, la empresa **Sun Microsystems** crea el lenguaje **Oak (**de la mano del llamado proyecto **Green).** Mediante este lenguaje se pretendía crear un sistema de televisión interactiva. Este lenguaje sólo se llegó a utilizar de forma interna. Su propósito era crear un lenguaje independiente de la plataforma y para uso en dispositivos electrónicos.

Se intentaba con este lenguaje paliar el problema fundamental del C++; que consiste en que al compilar se produce un fichero ejecutable cuyo código sólo vale para la plataforma en la que se realizó la compilación. Sun deseaba un lenguaje para programar

pequeños dispositivos electrónicos. La dificultad de estos dispositivos es que cambian continuamente y para que un programa funcione en el siguiente dispositivo aparecido, hay que rescribir el código. Por eso Sun quería crear un lenguaje **independiente del dispositivo.**

En 1995 pasa a llamarse **Java** y se da a conocer al público. Adquiere notoriedad rápidamente. Java pasa a ser un lenguaje totalmente independiente de la plataforma y a la vez potente y orientado a objetos. Esa filosofía y su facilidad para crear aplicaciones para redes TCP/IP ha hecho que sea uno de los lenguajes más utilizados en la actualidad. La versión actual de Java es el llamado Java 2. Sus ventajas sobre C++ son:

- Su sintaxis es similar a C y C++
- No hay punteros (lo que le hace más seguro)
- Totalmente orientado a objetos
- Muy preparado para aplicaciones TCP/IP
- Implementa excepciones de forma nativa
- Es interpretado (lo que acelera su ejecución remota, aunque provoca que las aplicaciones Java se ejecuten más lentamente que las C++ en un ordenador local).
- Permite multihilos
- Admite firmas digitales
- Tipos de datos y control de sintaxis más rigurosa
- Es independiente de la plataforma

La última ventaja (quizá la más importante) se consigue ya que el código Java no se compila, sino que se **precompila,** de tal forma que se crea un código intermedio que no es ejecutable. Para ejecutarle hace falta pasarle por un intérprete que va ejecutando cada línea. Ese intérprete suele ser la máquina virtual de Java,

Java y JavaScript

Una de las confusiones actuales la provoca el parecido nombre que tienen estos dos lenguajes. Sin embargo no tienen nada que ver entre sí; Sun creo Java y Netscape creo JavaScript. Java es un lenguaje completo que permite realizar todo tipo de aplicaciones. JavaScript es código que está inmerso en una página web.

La finalidad de JavaScript es mejorar el dinamismo de las páginas web. La finalidad de Java es crear aplicaciones de todo tipo (aunque está muy preparado para crear sobre todo aplicaciones en red). Finalmente la sintaxis de ambos lenguajes apenas se parece,

características de Java

bytecodes

Un programa C o C++ es totalmente ejecutable y eso hace que no sea independiente de la plataforma y que su tamaño normalmente se dispare ya que dentro del código final hay que incluir las librerías de la plataforma

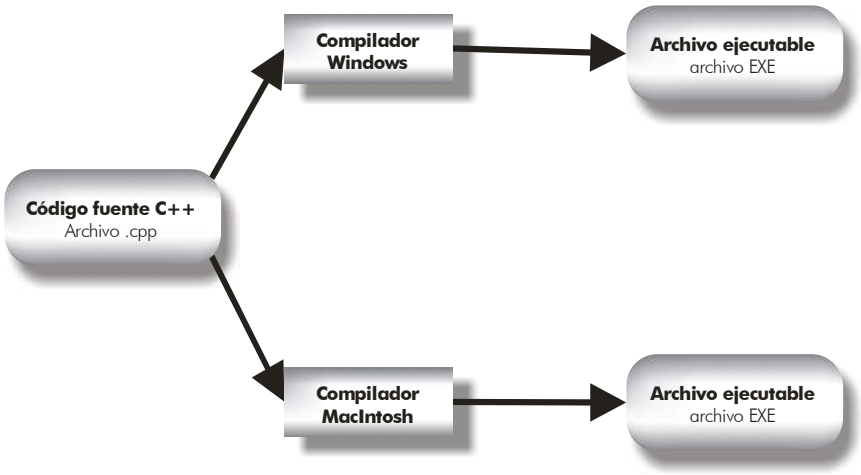


Ilustración 2, ***Proceso de compilación de un programa C++***

Los programas Java no son ejecutables, no se compilan como los programas en C o C++. En su lugar son interpretados por una aplicación conocida como la **máquina virtual de Java** (JVM). Gracias a ello no tienen porque incluir todo el código y librerías propias de cada sistema.

Previamente el código fuente en Java se tiene que precompilar generando un código (que no es directamente ejecutable) previo conocido como ***bytecode*** o ***J-code***. Ese código (generado normalmente en archivos con extensión **class**) es el que es ejecutado por la máquina virtual de Java que interpreta las instrucciones de los bytecodes, ejecutando el código de la aplicación.

El bytecode se puede ejecutar en cualquier plataforma, lo único que se requiere es que esa plataforma posea un intérprete adecuado (la máquina virtual de esa plataforma). La máquina virtual de Java, además es un programa muy pequeño y que se distribuye gratuitamente para prácticamente todos los sistemas operativos. A este método de ejecución de programas en tiempo real se le llama *Just in Time* (JIT).

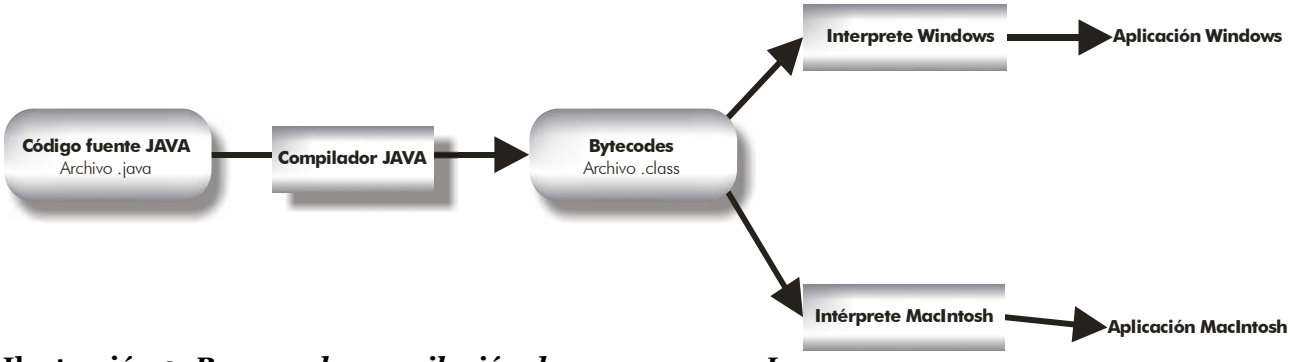


Ilustración 3, ***Proceso de compilación de un programa Java***

En Java la unidad fundamental del código es la **clase**. Son las clases las que se distribuyen en el formato *bytecode* de Java. Estas clases se cargan dinámicamente durante la ejecución del programa Java.

seguridad

Al interpretar el código, la JVM puede delimitar las operaciones peligrosas, con lo cual la seguridad es fácilmente controlable. Además, Java elimina las instrucciones dependientes de la máquina y los **punteros** que generaban terribles errores en C y la posibilidad de

generar programas para atacar sistemas. Tampoco se permite el acceso directo a memoria y además.

La primera línea de seguridad de Java es un **verificador del bytecode** que permite comprobar que el comportamiento del código es correcto y que sigue las reglas de Java. Normalmente los compiladores de Java no pueden generar código que se salte las reglas de seguridad de Java. Pero un programador *malévolo* podría generar artificialmente código *bytecode* que se salte las reglas. El verificador intenta eliminar esta posibilidad.

Hay un segundo paso que verifica la seguridad del código que es el **verificador de clase** que es el programa que proporciona las clases necesarias al código. Lo que hace es asegurarse que las clases que se cargan son realmente las del sistema original de Java y no clases creadas reemplazadas artificialmente.

Finalmente hay un **administrador de seguridad** que es un programa configurable que permite al usuario indicar niveles de seguridad a su sistema para todos los programas de Java.

Hay también una forma de seguridad relacionada con la confianza. Esto se basa es saber que el código Java procede de un sitio de confianza y no de una fuente no identificada. En Java se permite añadir firmas digitales al código para verificar al autor del mismo.

tipos de aplicaciones Java

applet

Son programas Java pensados para ser colocados dentro de una página web. Pueden ser interpretados por cualquier navegador con capacidades Java. Estos programas se insertan en las páginas usando una etiqueta especial (como también se insertan vídeos, animaciones flash u otros objetos).

Los applets son programas independientes, pero al estar incluidos dentro de una página web las reglas de éstas le afectan. Normalmente un applet sólo puede actuar sobre el navegador.

Hoy día mediante applets se pueden integrar en las páginas web aplicaciones multimedia avanzadas (incluso con imágenes 3D o sonido y vídeo de alta calidad)

aplicaciones de consola

Son programas independientes al igual que los creados con los lenguajes tradicionales.

aplicaciones gráficas

Aquellas que utilizan las clases con capacidades gráficas (como **awt** por ejemplo).

servlets

Son aplicaciones que se ejecutan en un servidor de aplicaciones web y que como resultado de su ejecución resulta una página web.

empezar a trabajar con Java

el kit de desarrollo Java (JDK)

Para escribir en Java hacen falta los programas que realizan el precompilado y la interpretación del código, Hay entornos que permiten la creación de los bytecodes y que incluyen herramientas con capacidad de ejecutar aplicaciones de todo tipo. El más famoso

(que además es gratuito) es el **Java Developer Kit (JDK)** de Sun, que se encuentra disponible en la dirección http://java.sun.com.

Actualmente ya no se le llama así sino que se le llama SDK y en la página se referencia la plataforma en concreto.

versiones de Java

Como se ha comentado anteriormente, para poder crear los bytecodes de un programa Java, hace falta el JDK de Sun. Sin embargo, Sun va renovando este kit actualizando el lenguaje. De ahí que se hable de Java 1.1, Java 1.2, etc.

Actualmente se habla de Java 2 para indicar las mejoras en la versión. Desde la versión 1.2 del JDK, el Kit de desarrollo se llama *Java 2 Developer Kit* en lugar de *Java Developer Kit*. La última versión es la 1.4.2.

Lo que ocurre (como siempre) con las versiones, es que para que un programa que utilice instrucciones del JDK 1.4.1, sólo funcionará si la máquina en la que se ejecutan los bytecodes dispone de un intérprete compatible con esa versión.

Java 1.0

Fue la primera versión de Java y propuso el marco general en el que se desenvuelve Java. está oficialmente obsoleto, pero hay todavía muchos clientes con esta versión.

Java 1.1

Mejoró la versión anterior incorporando las siguientes mejoras:

- El paquete **AWT** que permite crear interfaces gráficos de usuario, GUI.
- **JDBC** que es por ejemplo. Es soportado de forma nativa tanto por Internet Explorer como por Netscape Navigator.
- **RMI** llamadas a métodos remotos. Se utilizan por ejemplo para llamar a métodos de objetos alojados en servidor.
- Internacionalización para crear programas adaptables a todos los idiomas

Java 2

Apareció en Diciembre de 1998 al aparecer el JDK 1.2. Incorporó notables mejoras como por ejemplo:

- **JFC**. *Java Foundation classes.* El conjunto de clases de todo para crear programas más atractivos de todo tipo. Dentro de este conjunto están:
  - **El paquete Swing.** Sin duda la mejora más importante, este paquete permite realizar lo mismo que AWT pero superándole ampliamente.
  - **Java Media**
- **Enterprise Java beans.** Para la creación de componentes para aplicaciones distribuidas del lado del servidor
- **Java Media.** Conjunto de paquetes para crear paquetes multimedia:
  - **Java 2D.** Paquete (parte de JFC) que permite crear gráficos de alta calidad en los programas de Java.

- **Java 2D.** Paquete (parte de JFC) que permite crear gráficos tridimensionales.
- **Java Media Framework.** Paquete marco para elementos multimedia
- **Java Speech.** Reconocimiento de voz.
- **Java Sound.** Audio de alta calidad
- **Java TV.** Televisión interactiva

- **JNDI.** *Java Naming and Directory Interface*. Servicio general de búsqueda de recursos. Integra los servicios de búsqueda más populares (como LDAP por ejemplo).
- **Java Servlets.** Herramienta para crear aplicaciones de servidor web (y también otros tipos de aplicaciones).
- **Java Cryptography.** Algoritmos para encriptar.
- **Java Help.** Creación de sistemas de ayuda.
- **Jini.** Permite la programación de electrodomésticos.
- **Java card.** Versión de Java dirigida a pequeños dispositivos electrónicos.

Java 1.3 y 1.4

Son las últimas versiones de Java2 que incorporan algunas mejoras,

plataformas

Actualmente hay tres ediciones de la plataforma Java 2

J2SE

Se denomina así al entorno de Sun relacionado con la creación de aplicaciones y applets en lenguaje Java. la última versión del kit de desarrollo de este entorno es el J2SE 1.4.2.

J2EE

Pensada para la creación de aplicaciones Java empresariales y del lado del servidor. Su última versión es la 1.4

J2ME

Pensada para la creación de aplicaciones Java para dispositivos móviles.

entornos de trabajo

El código en Java se puede escribir en cualquier editor de texto. Y para compilar el código en bytecodes, sólo hace falta descargar la versión del JDK deseada. Sin embargo, la escritura y compilación de programas así utilizada es un poco incomoda. Por ello numerosas empresas fabrican sus propios entornos de edición, algunos incluyen el compilador y otras utilizan el propio JDK de Sun.

- NetBeans. Entorno gratuito de código abierto para la generación de código en diversos lenguajes (especialmente pensado para Java). Contiene prácticamente todo lo que se suele pedir a un IDE, editor avanzado de código, depurador, diversos

lenguajes, extensiones de todo tipo (CORBA, Servlets,...). Incluye además un servidor de aplicaciones Tomcat para probar aplicaciones de servidor. Se descarga en www.netbeans.org.

- Eclipse. Es un entorno completo de código abierto que admite numerosas extensiones (incluido un módulo para J2EE) y posibilidades. Es uno de los más utilizados por su compatibilidad con todo tipo de aplicaciones Java y sus interesantes opciones de ayuda al escribir código.

- Sun ONE Studio. Entorno para la creación de aplicaciones Java creado por la propia empresa Sun a partir de NetBeans (casi es clavado a éste). la versión *Community Edition* es gratuita (es más que suficiente), el resto son de pago. Está basado en el anterior. Antes se le conocía con el nombre Forte for Java. Está implicado con los servidores ONE de Java.

- Microsoft Visual J++ y Visual J#. Ofrece un compilador. El más recomendable para los conocedores de los editores y compiladores de Microsoft (como Visual Basic por ejemplo) aunque el Java que edita está más orientado a las plataformas de servidor de Microsoft.

- Visual Cafe. Otro entorno veterano completo de edición y compilado. Bastante utilizado. Es un producto comercial de la empresa Symantec.

- JBuilder. Entorno completo creado por la empresa Borland (famosa por su lenguaje Delphi) para la creación de todo tipo de aplicaciones Java, incluidas aplicaciones para móviles.

- JDeveloper. De Oracle. Entorno completo para la construcción de aplicaciones Java y XML. Uno de los más potentes y completos (ideal para programadores de Oracle).

- Visual Age. Entorno de programación en Java desarrollado por IBM. Es de las herramientas más veteranas. Actualmente en desuso.

- IntelliJ Idea. Entorno comercial de programación bastante fácil de utilizar pero a la vez con características similares al resto. Es menos pesado que los anteriores y muy bueno con el código.

- JCreator Pro. Es un editor comercial muy potente y de precio bajo. Ideal (junto con Kawa) para centrarse en el código Java. No es un IDE completo y eso lo hace más ligero, de hecho funciona casi en cualquier máquina.

- Kawa Pro. Muy similar al anterior. Actualmente se ha dejado de fabricar.

escritura de programas Java

codificación del texto

Todos el código fuente Java se escriben en documentos de texto con extensión **.java**. Al ser un lenguaje para Internet, la codificación de texto debía permitir a todos los programadores de cualquier idioma escribir ese código. Eso significa que Java es compatible con la codificación Unicode.

En la práctica significa que los programadores que usen lenguajes distintos del inglés no tendrán problemas para escribir símbolos de su idioma. Y esto se puede extender para nombres de clase, variables, etc.

La codificación Unicode[2] usa 16 bits (2 bytes por carácter) e incluye la mayoría de los códigos del mundo.

notas previas

Los archivos con código fuente en Java deben guardarse con la extensión .java. Como se ha comentado cualquier editor de texto basta para crearle. Algunos detalles importantes son:

- En java (como en C) hay diferencia entre mayúsculas y minúsculas.
- Cada línea de código debe terminar con ;
- Los comentarios; si son de una línea debe comenzar con "//" y si ocupan más de una línea deben comenzar con "/*" y terminar con "*/"

```
/* Comentario
de varias líneas */
//Comentario de una línea
```

También se pueden incluir comentarios **javadoc** (ver más adelante)

- A veces se marcan bloques de código, los cuales comienza con { y terminan con } El código dentro de esos símbolos se considera interno al bloque

```
{
   ...código dentro del bloque
}
código fuera del bloque
```

el primer programa en Java

```
public class app
{
  public static void main(String[] args)
  {
        System.out.println("¡Mi primer programa!");
  }
}
```

Este código escribe "¡Mi primer programa!" en la pantalla. El archivo debería llamarse **app.java** ya que esa es la clase pública. El resto define el método **main** que es el que se ejecutará al lanzarse la aplicación. Ese método utiliza la instrucción que escribe en pantalla.

proceso de compilación

Hay que entender que Java es estricto en cuanto a la interpretación de la programación orientada a objetos. Así, se sobrentiende que un archivo java crea una (y sólo) clase. Por eso al compilar se dice que lo que se está compilando es una clase.

[2] Para más información acudir a http://www.unicode.org

La compilación del código java se realiza mediante el programa **javac** incluido en el software de desarrollo de java. La forma de compilar es (desde la línea de comandos):

```
javadoc archivo.java
```

El resultado de esto es un archivo con el mismo nombre que el archivo java pero con la extensión **class**. Esto ya es el archivo con el código en forma de **bytecodes**. Es decir con el código precompilado.

Si la clase es ejecutable (sólo lo son si contienen el método **main**), el código se puede interpretar usando el programa **java** del kit de desarrollo. Sintaxis:

```
java archivoClass
```

Estos comandos hay que escribirlos desde la línea de comandos de en la carpeta en la que se encuentre el programa. Pero antes hay que asegurarse de que los programas del kit de desarrollo son accesibles desde cualquier carpeta del sistema. Para ello hay que comprobar que la carpeta con los ejecutables del kit de desarrollo está incluida en la variable de entorno **path.**

Esto lo podemos comprobar escribiendo **path** en la línea de comandos. Si la carpeta del kit de desarrollo no está incluida, habrá que hacerlo. Para ello en Windows 2000 o XP:

1> Pulsar el botón derecho sobre Mi PC y elegir **Propiedades**

2> Ir al apartado **Opciones avanzadas**

3> Hacer clic sobre el botón **Variables de entorno**

4> Añadir a la lista de la variable **Path** la ruta a la carpeta con los programas del JDK.

Ejemplo de contenido de la variable path:

```
PATH=C:\WINNT\SYSTEM32;C:\WINNT;C:\WINNT\SYSTEM32\WBEM;C:\Arch
ivos de programa\Microsoft Visual
Studio\Common\Tools\WinNT;C:\Archivos de programa\Microsoft
Visual Studio\Common\MSDev98\Bin;C:\Archivos de
programa\Microsoft Visual Studio\Common\Tools;C:\Archivos de
programa\Microsoft Visual Studio\VC98\bin;C:\Archivos de
programa\j2sdk_nb\j2sdk1.4.2\bin
```

En negrita está señalada la ruta a la carpeta de ejecutables (carpeta bin) del kit de desarrollo. Está carpeta varía según la instalación

javadoc

Javadoc es una herramienta muy interesante del kit de desarrollo de Java para generar automáticamente documentación Java. genera documentación para paquetes completos o para archivos java. Su sintaxis básica
es:

```
javadoc archivo.java o paquete
```

El funcionamiento es el siguiente. Los comentarios que comienzan con los códigos /** se llaman comentarios de documento y serán utilizados por los programas de generación de documentación javadoc.

Los comentarios javadoc comienzan con el símbolo /** y terminan con */ Cada línea javadoc se inicia con un símbolo de asterisco. Dentro se puede incluir cualquier texto. Incluso se pueden utilizar códigos HTML para que al generar la documentación se tenga en cuenta el código HTML indicado.

En el código javadoc se pueden usar **etiquetas** especiales, las cuales comienzan con el símbolo @. Pueden ser:

- **@author.** Tras esa palabra se indica el autor del documento.
- **@version.** Tras lo cual sigue el número de versión de la aplicación
- **@see.** Tras esta palabra se indica una referencia a otro código Java relacionado con éste.
- **@since.** Indica desde cuándo esta disponible este código
- **@deprecated.** Palabra a la que no sigue ningún otro texto en la línea y que indica que esta clase o método esta obsoleta u obsoleto.
- **@throws.** Indica las excepciones que pueden lanzarse en ese código.
- **@param.** Palabra a la que le sigue texto qué describe a los parámetros que requiere el código para su utilización (el código en este caso es un método de clase). Cada parámetro se coloca en una etiqueta @param distinta, por lo que puede haber varios @param para el mismo método.
- **@return.** Tras esta palabra se describe los valores que devuelve el código (el código en este caso es un método de clase)

El código javadoc hay que colocarle en tres sitios distintos dentro del código java de la aplicación:

1> **Al principio del código de la clase** (antes de cualquier código Java). En esta zona se colocan comentarios generales sobre la clase o interfaz que se crea mediante el código Java. Dentro de estos comentarios se pueden utilizar las etiquetas: @author, @version, @see, @since y @deprecated

2> **Delante de cada método.** Los métodos describen las cosas que puede realizar una clase. Delante de cada método los comentarios javadoc se usan para describir al método en concreto. Además de los comentarios, en esta zona

se pueden incluir las etiquetas: @see, @param, @exception, @return, @since y @deprecated

3> **Delante de cada atributo.** Se describe para qué sirve cada atributo en cada clase. Puede poseer las etiquetas: @since y @deprecated

Ejemplo:

```
/** Esto es un comentario para probar el javadoc
* este texto aparecerá en el archivo HTML generado.
* <strong>Realizado en agosto 2003</strong>
*
* @author Jorge Sánchez
* @version 1.0
*/
public class prueba1 {
//Este comentario no aparecerá en el javadoc

  /** Este método contiene el código ejecutable de la clase
  *
  * @param args Lista de argumentos de la línea de comandos
  * @return void
  */

  public static void main(String args[]){
        System.out.println("¡Mi segundo programa! ");
  }
}
```

Tras ejecutar la aplicación **javadoc,** aparece como resultado la página web de la página siguiente.

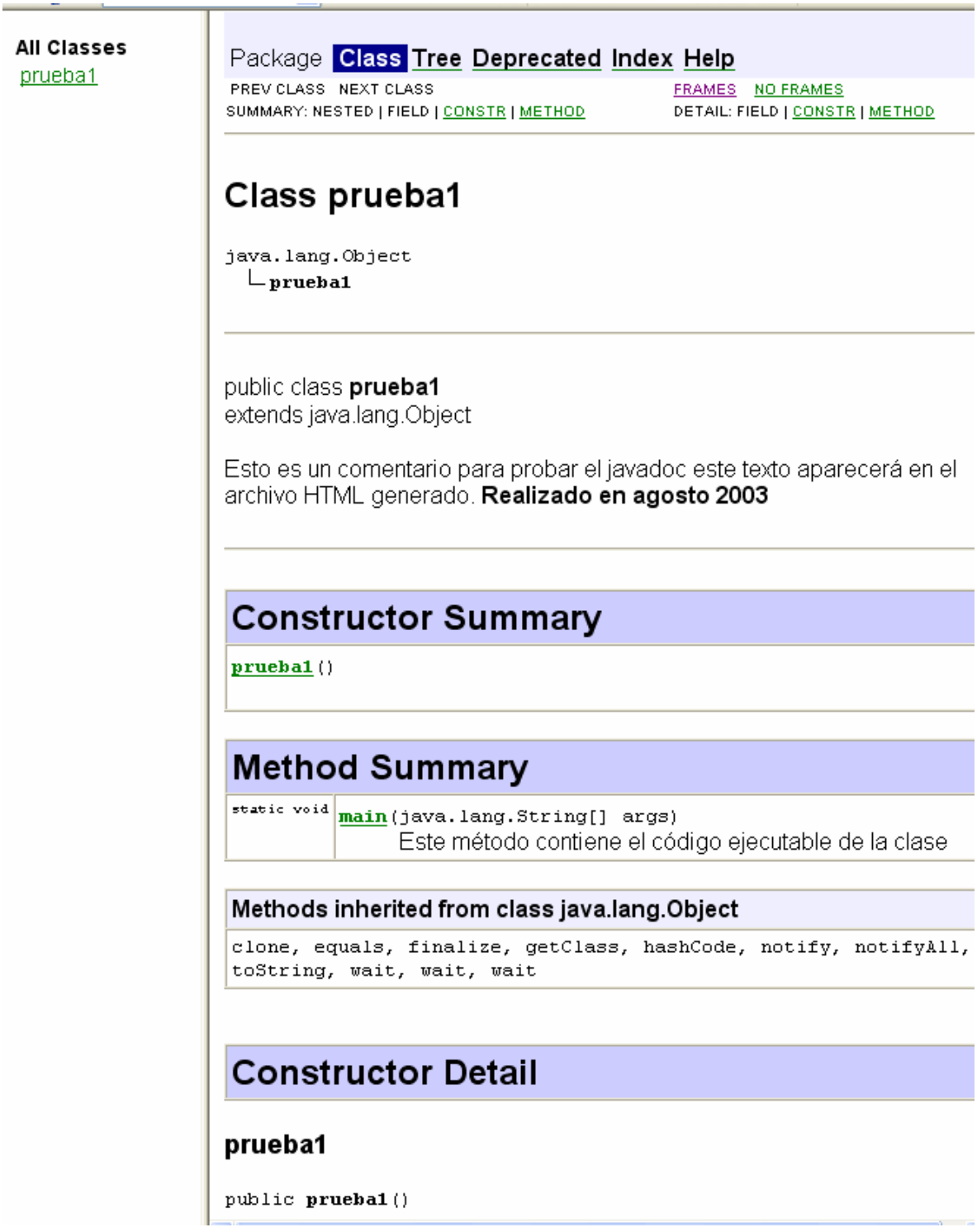


Ilustración 4, Página de documentación de un pro grama Java

instrucción import

Hay código que se puede utilizar en los programas que realicemos en Java. Se importar clases de objetos que están contenidas, a su vez, en paquetes estándares.

Por ejemplo la clase **Date** es una de las más utilizadas, sirve para manipular fechas. Si alguien quisiera utilizar en su código objetos de esta clase, necesita incluir una instrucción que permita utilizar esta clase. La sintaxis de esta instrucción es:

```
import paquete.subpaquete.subsubapquete....clase
```

Esta instrucción se coloca arriba del todo en el código. Para la clase **Date** sería:

```
import java.util.Date
```

Lo que significa, importar en el código la clase **Date** que se encuentra dentro del paquete **util** que, a su vez, está dentro del gran paquete llamado **java**.

También se puede utilizar el asterisco en esta forma:

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

Esto significa que se va a incluir en el código todas las clases que están dentro del paquete **util** de **java.**

variables

introducción

Las variables son los contenedores de los datos que utiliza un programa. Cada variable ocupa un espacio en la memoria RAM del ordenador para almacenar un dato determinado.

Las variables tienen un nombre (un **identificador**) que sólo puede contener letras, números y el carácter de subrayado (también vale el símbolo $). El nombre puede contener cualquier carácter Unicode.

declaración de variables

Antes de poder utilizar una variable, ésta se debe declarar. Lo cual se debe hacer de esta forma:

```
tipo nombrevariable;
```

Donde ***tipo*** es el tipo de datos que almacenará la variable (texto, números enteros,...) y *nombrevariable* es el nombre con el que se conocerá la variable. Ejemplos:

```
int dias;
boolean decision;
```

También se puede hacer que la variable tome un valor inicial al declarar:

```
int dias=365;
```

Y también se puede declarar más de una variable a la vez:

```
int dias=365, anio=23, semanas;
```

Al declarar una variable se puede incluso utilizar una expresión:

```
int a=13, b=18;
int c=a+b;
```

alcance o ámbito

Esas dos palabras sinónimas, hacen referencia a la duración de una variable. En el ejemplo:

```
{
  int x=12;
}
System.out.println(x); //Error
```

Java dará error, porque la variable se usa fuera del bloque en el que se creo. Eso no es posible, porque una variable tiene como ámbito el bloque de código en el que fue creada (salvo que sea una propiedad de un objeto).

tipos de datos primitivos

Tipo de variable	Bytes que ocupa	Rango de valores
boolean	**2**	true, false
byte	**1**	**-128 a 127**
short	**2**	**-32.768 a 32.767**
int	**4**	**-2.147.483.648 a 2.147.483.649**
long	**8**	**$-9 \cdot 10^{18}$ a $9 \cdot 10^{18}$**
double	**8**	**$-1{,}79 \cdot 10^{308}$ a $1{,}79 \cdot 10^{308}$**
float	**4**	**$-3{,}4 \cdot 10^{38}$ a $3{,}4 \cdot 10^{38}$**
char	**2**	**Caracteres (en Unicode)**

enteros

Los tipos **byte, short, int** y **long** sirven para almacenar datos enteros. Los enteros son números sin decimales. Se pueden asignar enteros normales o enteros octales y hexadecimales. Los octales se indican anteponiendo un cero al número, los hexadecimales anteponiendo 0x.

```
int numero=16; //16 decimal
numero=020; //20 octal=16 decimal
numero=0x14; //10 hexadecimal=16 decimal
```

Normalmente un número literal se entiende que es de tipo **int** salvo si al final se le coloca la letra L; se entenderá entonces que es de tipo long.

No se acepta en general asignar variables de distinto tipo. Sí se pueden asignar valores de variables enteras a variables enteras de un tipo superior (por ejemplo asignar un valor **int** a una variable **long**). Pero al revés no se puede:

```
int i=12;
byte b=i;     //error de compilación
```

La solución es hacer un **cast**. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i=12;
byte b=(byte) i;   //No hay problema por el (cast)
```

Hay que tener en cuenta en estos *cast* que si el valor asignado sobrepasa el rango del elemento, el valor convertido no tendrá ningún sentido:

```
int i=1200;
byte b=(byte) i;   //El valor de b no tiene sentido
```

números en coma flotante

Los decimales se almacenan en los tipos **float** y **double**. Se les llama de coma flotante por como son almacenados por el ordenador. Los decimales no son almacenados de forma exacta por eso siempre hay un posible error. En los decimales de coma flotante se habla, por tanto de precisión. Es mucho más preciso el tipo **double** que el tipo **float.**

A un valor literal (como 1.5 por ejemplo), se le puede indicar con una **f** al final del número que es float (1.5f por ejemplo) o una **d** para indicar que es **double.** Si no se indica nada, un número literal siempre se entiende que es double, por lo que al usar tipos float hay que convertir los literales.

Las valores decimales se pueden representar en notación decimal: 1.345E+3 significaría $1.345 \cdot 10^3$ o lo que es lo mismo 1345.

booleanos

Los valores booleanos (o lógicos) sirven para indicar si algo es verdadero (**true**) o falso (**false**). En C se puede utilizar cualquier valor lógico como si fuera un número; así verdadero es el valor -1 y falso el 0. Eso no es posible en Java.

Si a un valor booleano no se le da un valor inicial, se toma como valor inicial el valor **false.** Por otro lado, a diferencia del lenguaje C, no se pueden en Java asignar números a una variable booleana (en C, el valor false se asocia al número 0, y cualquier valor distinto de cero se asocia a true).

caracteres

Los valores de tipo carácter sirven para almacenar símbolos de escritura (en Java se puede almacenar cualquier código Unicode). Los valores Unicode son los que Java utiliza para los caracteres. Ejemplo:

```
char letra;
letra='C'; //Los caracteres van entre comillas
letra=67; //El código Unicode de la C es el 67. Esta línea
            //hace lo mismo que la anterior
```

También hay una serie de caracteres especiales que van precedidos por el símbolo \, son estos:

carácter	significado
\b	Retroceso
\t	Tabulador
\n	Nueva línea
\f	Alimentación de página
\r	Retorno de carro

carácter	significado
\”	Dobles comillas
\’	Comillas simples
udddd	Las cuatro letras d, son en realidad números en hexadecimal. Representa el carácter Unicode cuyo código es representado por las *dddd*

conversión entre tipos *(casting)*

Hay veces en las que se deseará realizar algo como:

```
int a;byte b=12;
a=b;
```

La duda está en si esto se puede realizar. La respuesta es que sí. Sí porque un dato byte es más pequeño que uno int y Java le convertirá de forma implícita. Sin embargo en:

```
int a=1;
byte b;
b=a;
```

El compilador devolverá error aunque el número 1 sea válido para un dato byte. Para ello hay que hacer un *casting*. Eso significa poner el tipo deseado entre paréntesis delante de la expresión.

```
int a=1;
byte b;
b= (byte) a; //No da error
```

En el siguiente ejemplo:

```
byte n1=100, n2=100, n3;
n3= n1 * n2 /100;
```

Aunque el resultado es 100, y ese resultado es válido para un tipo byte; lo que ocurrirá en realidad es que ocurrirá un error. Eso es debido a que primero multiplica 100 * 100 y como eso da 10000, no tiene más remedio el compilador que pasarlo a entero y así quedará aunque se vuelva a dividir. La solución correcta sería:

```
n3 = (byte) (n1 * n2 / 100);
```

operadores

introducción

Los datos se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, ... y a veces se realizan operaciones más complejas.

operadores aritméticos

Son:

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (resto)

Hay que tener en cuenta que el resultado de estos operadores varía notablemente si usamos enteros o si usamos números de coma flotante.

Por ejemplo:

```
double resultado1, d1=14, d2=5;
int resultado2, i1=14, i2=5;

resultado1= d1 / d2;
resultado2= i1 / i2;
```

resultado1 valdrá 2.8 mientras que *resultado2* valdrá 2. Es más incluso:

```
double resultado;
int i1=7,i2=2;
resultado=i1/i2; //Resultado valdrá 3
resultado=(double)i1/(double)i2; //Resultado valdrá 3.5
```

El operador del módulo (%) para calcular el resto de una división entera. Ejemplo:

```
int resultado, i1=14, i2=5;

resultado = i1 % i2; //El resultado será 4
```

operadores condicionales

Sirven para comparar valores. Siempre devuelven valores booleanos. Son:

operador	significado
<	Menor
>	Mayor
>=	Mayor o igual
<=	Menor o igual
==	Igual

operador	significado
!=	Distinto
!	No lógico (NOT)
&&	"Y" lógico (AND)
||	"O" lógico (OR)

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
int edad = 21;
mayorDeEdad = edad >= 18; //mayorDeEdad será true
menorDeEdad = !mayorDeEdad; //menorDeEdad será false
```

El operador && (AND) sirve para evaluar dos expresiones de modo que si ambas son ciertas, el resultado será **true** sino el resultado será **false**. Ejemplo:

```
boolean carnetConducir=true;
int edad=20;
boolean puedeConducir= (edad>=18) && carnetConducir;
//Si la edad es de al menos 18 años y carnetConducir es
//true, puedeConducir es true
```

El operador || (OR) sirve también para evaluar dos expresiones. El resultado será **true** si al menos uno de las expresiones es **true**. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;
boolean malTiempo= nieva || llueve || graniza;
```

operadores de BIT

Manipulan los bits de los números. Son:

operador	significado
&	AND
|	OR
~	NOT
^	XOR
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda
>>>	Desplazamiento derecha con relleno de ceros
<<<	Desplazamiento izquierda con relleno de ceros

operadores de asignación

Permiten asignar valores a una variable. El fundamental es "=". Pero sin embargo se pueden usar expresiones más complejas como:

```
x += 3;
```

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo x+=3, que x=x+3). Eso se puede hacer también con todos estos operadores:

+=	-=	*=	/=
&=	|=	^=	%=
>>=	<<=		

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5;
```

Otros operadores de asignación son "++" (incremento) y "- -" (decremento). Ejemplo:

```
x++; //esto es x=x+1;
x--; //esto es x=x-1;
```

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo x++ o ++x

La diferencia estriba en el modo en el que se comporta la asignación. Ejemplo:

```
int x=5, y=5, z;
z=x++; //z vale 5, x vale 6
z=++y; //z vale 6, y vale 6
```

operador ?

Este operador (conocido como **if** de una línea) permite ejecutar una instrucción u otra según el valor de la expresión. Sintaxis:

```
expresionlogica?valorSiVerdadero:valorSiFalso;
```

Ejemplo:

```
paga=(edad>18)?6000:3000;
```

En este caso si la variable edad es mayor de 18, la paga será de 6000, sino será de 3000. Se evalúa una condición y según es cierta o no se devuelve un valor u otro. Nótese que esta función ha de devolver un valor y no una expresión correcta. Es decir, no funcionaría:

```
(edad>18)? paga=6000: paga=3000; /ERROR!!!!
```

precedencia

A veces hay expresiones con operadores que resultan confusas. Por ejemplo en:

```
resultado = 8 + 4 / 2;
```

Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución al de la suma. Es decir, siempre se ejecuta antes la división que la suma. Siempre se pueden usar paréntesis para forzar el orden deseado:

```
resultado = (8 + 4) / 2;
```

Ahora no hay duda, el resultado es seis. No obstante el orden de precedencia de los operadores Java es:

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

En la tabla anterior los operadores con mayor precedencia está en la parte superior, los de menor precedencia en la parte inferior. De izquierda a derecha la precedencia es la misma. Es decir, tiene la misma precedencia el operador de suma que el de resta.

Esto último provoca conflictos, por ejemplo en:

```
resultado = 9 / 3 * 3;
```

El resultado podría ser uno ó nueve. En este caso el resultado es nueve, porque la división y el producto tienen la misma precedencia; por ello el compilador de Java realiza primero la operación que este más a la izquierda, que en este caso es la división.

Una vez más los paréntesis podrían evitar estos conflictos.

la clase Math

Se echan de menos operadores matemáticos más potentes en Java. Por ello se ha incluido una clase especial llamada **Math** dentro del paquete **java.lang.**Para poder utilizar esta clase, se debe incluir esta instrucción:

```
import java.lang.Math;
```

Esta clase posee métodos muy interesantes para realizar cálculos matemáticos complejos. Por ejemplo:

```
double x= Math.pow(3,3); //x es 3^3
```

Math posee dos constantes, que son:

constante	significado
double E	El número **e** (2, 7182818245...)
double PI	El número **∏** (3,14159265...)

Por otro lado posee numerosos métodos que son:

operador	significado
double ceil(double *x*)	Redondea *x* al entero mayor siguiente: ◉ Math.ceil(2.8) vale 3 ◉ Math.ceil(2.4) vale 3 ◉ Math.ceil(-2.8) vale -2
double floor(double *x*)	Redondea *x* al entero menor siguiente: ◉ Math.floor(2.8) vale 2 ◉ Math. floor (2.4) vale 2 ◉ Math. floor (-2.8) vale -3
int round(double *x*)	Redondea *x* de forma clásica: ◉ Math.round(2.8) vale 3 ◉ Math. round (2.4) vale 2 ◉ Math. round (-2.8) vale -3
double rint(double *x*)	Idéntico al anterior, sólo que éste método da como resultado un número **double** mientras que **round** da como resultado un entero tipo **int**
double random()	Número aleatorio de 0 a 1
***tiponúmero* abs(*tiponúmero* *x*)**	Devuelve el valor absoluto de *x*.

operador	significado
tiponúmero **min(** ***tiponúmero*** *x,* ***tiponúmero*** *y***)**	Devuelve el menor valor de *x* o *y*
tiponúmero **max(** ***tiponúmero*** *x,* ***tiponúmero*** *y***)**	Devuelve el mayor valor de *x* o *y*
double sqrt(double *x***)**	Calcula la raíz cuadrada de *x*
double pow(double *x***, double** *y)*	Calcula x^y
double exp(double *x***)**	Calcula e^x
double log(double *x***)**	Calcula el logaritmo neperiano de *x*
double acos(double *x***)**	Calcula el arco coseno de *x*
double asin(double *x***)**	Calcula el arco seno de *x*
double atan(double *x***)**	Calcula el arco tangente de *x*
double sin(double *x***)**	Calcula el seno de *x*
double cos(double *x***)**	Calcula el coseno de *x*
double tan(double *x***)**	Calcula la tangente de *x*
double toDegrees(double *anguloEnRadianes***)**	Convierte de radianes a grados
double toRadians(double *anguloEnGrados***)**	Convierte de grados a radianes

estructuras de control del flujo

if

Permite crear estructuras condicionales simples; en las que al cumplirse una condición se ejecutan una serie de instrucciones. Se puede hacer que otro conjunto de instrucciones se ejecute si la condición es falsa. La condición es cualquier expresión que devuelva un resultado de **true** o **false.** La sintaxis de la instrucción **if** es:

```
if (condición) {
   instrucciones que se ejecutan si la condición es true
}
else {
   instrucciones que se ejecutan si la condición es false
}
```

La parte **else** es opcional. Ejemplo:

```
if ((diasemana>=1) && (diasemana<=5)){
   trabajar = true;
}
else {
   trabajar = false;
}
```

Se pueden anidar varios if a la vez. De modo que se comprueban varios valores. Ejemplo:

```
if (diasemana==1) dia="Lunes";
else if (diasemana==2) dia="Martes";
else if (diasemana==3) dia="Miércoles";
else if (diasemana==4) dia="Jueves";
else if (diasemana==5) dia="Viernes";
else if (diasemana==6) dia="Sábado";
else if (diasemana==7) dia="Domingo";
else dia="?";
```

switch

Es la estructura condicional compleja porque permite evaluar varios valores a la vez. Sintaxis:

```
switch (expresión) {
   case valor1:
        sentencias si la expresiona es igual al valor1;
```

```
        [break]
  case valor2:
        sentencias si la expresiona es igual al valor2;
        [break]
        .
        .
        .
  default:
        sentencias que se ejecutan si no se cumple ninguna
        de las anteriores
}
```

Esta instrucción evalúa una expresión (que debe ser **short, int, byte** o **char**), y según el valor de la misma ejecuta instrucciones. Cada **case** contiene un valor de la expresión; si efectivamente la expresión equivale a ese valor, se ejecutan las instrucciones de ese **case** y de los siguientes.

La instrucción **break** se utiliza para salir del **switch**. De tal modo que si queremos que para un determinado valor se ejecuten las instrucciones de un apartado **case** y sólo las de ese apartado, entonces habrá que finalizar ese **case** con un **break.**

El bloque **default** sirve para ejecutar instrucciones para los casos en los que la expresión no se ajuste a ningún **case.**

Ejemplo 1:

```
switch (diasemana) {
  case 1:
        dia="Lunes";
        break;
  case 2:
        dia="Martes";
        break;
  case 3:
        dia="Miércoles";
        break;
  case 4:
        dia="Jueves";
        break;
  case 5:
        dia="Viernes";
        break;
  case 6:
        dia="Sábado";
        break;
  case 7:
        dia="Domingo";
```

```
        break;
  default:
        dia="?";
}
```

Ejemplo 2:

```
switch (diasemana) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
        laborable=true;
        break;
  case 6:
  case 7:
        laborable=false;
}
```

while

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten si se cumple una determinada condición. Los bucles **while** agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.

La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while

Sintaxis:

```
while (condición) {
  sentencias que se ejecutan si la condición es true
}
```

Ejemplo (cálculo del factorial de un número, el factorial de 4 sería: 4*3*2*1):

```
//factorial de 4
int n=4, factorial=1, temporal=n;

while (temporal>0) {
  factorial*=temporal--;
}
```

do while

Crea un bucle muy similar al anterior, en la que también las instrucciones del bucle se ejecutan hasta que una condición pasa a ser falsa. La diferencia estriba en que en este tipo de bucle la condición se evalúa después de ejecutar las instrucciones; lo cual significa que al menos el bucle se ejecuta una vez. Sintaxis:

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

for

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for (expresiónInicial; condición; expresiónEncadavuelta) {
   instrucciones;
}
```

La **expresión inicial** es una instrucción que se ejecuta una sola vez: al entrar por primera vez en el bucle **for** (normalmente esa expresión lo que hace es dar valor inicial al contador del bucle).

La **condición** es cualquier expresión que devuelve un valor lógico. En el caso de que esa expresión sea verdadera se ejecutan las instrucciones. Cuando la condición pasa a ser falsa, el bucle deja de ejecutarse. La condición se valora cada vez que se terminan de ejecutar las instrucciones del bucle.

Después de ejecutarse las instrucciones interiores del bucle, se realiza la expresión que tiene lugar tras ejecutarse las instrucciones del bucle (que, generalmente, incrementa o decrementa al contador). Luego se vuelve a evaluar la condición y así sucesivamente hasta que la condición sea falsa.

Ejemplo (factorial):

```
//factorial de 4
int n=4, factorial=1, temporal=n;

for (temporal=n;temporal>0;temporal--){
   factorial *=temporal;
}
```

sentencias de salida de un bucle

break

Es una sentencia que permite salir del bucle en el que se encuentra inmediatamente. Hay que intentar evitar su uso ya que produce malos hábitos al programar.

continue

Instrucción que siempre va colocada dentro de un bucle y que hace que el flujo del programa ignore el resto de instrucciones del bucle; dicho de otra forma, va hasta la siguiente iteración del bucle. Al igual que ocurría con **break**, hay que intentar evitar su uso.

arrays y cadenas

arrays

unidimensionales

Un array es una colección de valores de un mismo tipo engrosados en la misma variable. De forma que se puede acceder a cada valor independientemente. Para Java además un array es un objeto que tiene propiedades que se pueden manipular.

Los arrays solucionan problemas concernientes al manejo de muchas variables que se refieren a datos similares. Por ejemplo si tuviéramos la necesidad de almacenar las notas de una clase con 18 alumnos, necesitaríamos 18 variables, con la tremenda lentitud de manejo que supone eso. Solamente calcular la nota media requeriría una tremenda línea de código. Almacenar las notas supondría al menos 18 líneas de código.

Gracias a los arrays se puede crear un conjunto de variables con el mismo nombre. La diferencia será que un número (índice del array) distinguirá a cada variable.

En el caso de las notas, se puede crear un array llamado notas, que representa a todas las notas de la clase. Para poner la nota del primer alumno se usaría notas[0], el segundo sería notas[1], etc. (los corchetes permiten especificar el índice en concreto del array).

La declaración de un array unidimensional se hace con esta sintaxis.

```
tipo nombre[];
```

Ejemplo:

```
double cuentas[]; //Declara un array que almacenará valores
                  // doubles
```

Declara un array de tipo double. Esta declaración indica para qué servirá el array, pero no reserva espacio en la RAM al no saberse todavía el tamaño del mismo.

Tras la declaración del array, se tiene que iniciar. Eso lo realiza el operador **new**, que es el que realmente crea el array indicando un tamaño. Cuando se usa new es cuando se reserva el espacio necesario en memoria. Un array no inicializado es un array **null.** Ejemplo:

```
int notas[]; //sería válido también int[] notas;
notas = new int[3]; //indica que el array constará de tres
                    //valores de tipo int

//También se puede hacer todo a la vez
//int notas[]=new int[3];
```

En el ejemplo anterior se crea un array de tres enteros (con los tipos básicos se crea en memoria el array y se inicializan los valores, los números se inician a 0).

Los valores del array se asignan utilizando el índice del mismo entre corchetes:

```
notas[2]=8;
```

También se pueden asignar valores al array en la propia declaración:

```
int notas[] = {8, 7, 9};
int notas2[]= new int[] {8,7,9};//Equivalente a la anterior
```

Esto declara e inicializa un array de tres elementos. En el ejemplo lo que significa es que notas[0] vale 8, notas[1] vale 7 y notas[2] vale 9.

En Java (como en otros lenguajes) el primer elemento de un array es el cero. El primer elemento del array notas, es notas[0]. Se pueden declarar arrays a cualquier tipo de datos (enteros, booleanos, doubles, ... e incluso objetos).

La ventaja de usar arrays (volviendo al caso de las notas) es que gracias a un simple bucle **for** se puede rellenar o leer fácilmente todos los elementos de un array:

```
//Calcular la media de las 18 notas
suma=0;
for (int i=0;i<=17;i++){
   suma+=nota[i];
}
media=suma/18;
```

A un array se le puede inicializar las veces que haga falta:

```
int notas[]=new notas[16];
...
notas=new notas[25];
```

Pero hay que tener en cuenta que el segundo new hace que se pierda el contenido anterior. Realmente un array es una referencia a valores que se almacenan en memoria mediante el operador new, si el operador **new** se utiliza en la misma referencia, el anterior contenido se queda sin referencia y, por lo tanto se pierde.

Un array se puede asignar a otro array (si son del mismo tipo):

```
int notas[];
int ejemplo[]=new int[18];
notas=ejemplo;
```

En el último punto, notas equivale a ejemplo. Esta asignación provoca que cualquier cambio en notas también cambie el array ejemplos. Es decir esta asignación anterior, no copia los valores del array, sino que notas y ejemplo son referencias al mismo array. Ejemplo:

```
int notas[]={3,3,3};
int ejemplo[]=notas;
ejemplo= notas;
```

```
ejemplo[0]=8;
System.out.println(notas[0]);//Escribirá el número 8
```

arrays multidimensionales

Los arrays además pueden tener varias dimensiones. Entonces se habla de arrays de arrays (arrays que contienen arrays) Ejemplo:

```
int notas[][];
```

notas es un array que contiene arrays de enteros

```
notas = new int[3][12];//notas está compuesto por 3 arrays
                       //de 12 enteros cada uno
notas[0][0]=9;//el primer valor es 0
```

Puede haber más dimensiones incluso (notas[3][2][7]). Los arrays multidimensionales se pueden inicializar de forma más creativa incluso. Ejemplo:

```
int notas[][]=new int[5][];//Hay 5 arrays de enteros
notas[0]=new int[100]; //El primer array es de 100 enteros
notas[1]=new int[230]; //El segundo de 230
notas[2]=new int[400];
notas[3]=new int[100];
notas[4]=new int[200];
```

Hay que tener en cuenta que en el ejemplo anterior, notas[0] es un array de 100 enteros. Mientras que notas, es un array de 5 arrays de enteros.

Se pueden utilizar más de dos dimensiones si es necesario.

longitud de un array

Los arrays poseen un método que permite determinar cuánto mide un array. Se trata de **length.** Ejemplo (continuando del anterior):

```
System.out.println(notas.length); //Sale 5
System.out.println(notas[2].length); //Sale 400
```

la clase Arrays

En el paquete **java.utils** se encuentra una clase estática llamada **Arrays**. Una clase estática permite ser utilizada como si fuera un objeto (como ocurre con **Math**). Esta clase posee métodos muy interesantes para utilizar sobre arrays.

Su uso es

```
Arrays.método(argumentos);
```

fill

Permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

```
int valores[]=new int[23];
Arrays.fill(valores,-1);//Todo el array vale -1
```

También permite decidir desde que índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1);//Del elemento 5 al 7 valdrán -1
```

equals

Compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores.

sort

Permite ordenar un array en orden ascendente. Se pueden ordenar sólo una serie de elementos desde un determinado punto hasta un determinado punto.

```
int x[]={4,5,2,3,7,8,2,3,9,5};
Arrays.sort(x);//Estará ordenado
Arrays.sort(x,2,5);//Ordena del 2° al 4° elemento
```

binarySearch

Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está colocado el elemento. Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};
Arrays.sort(x);
System.out.println(Arrays.binarySearch(x,8));//Da  7
```

el método System.arraysCopy

La clase System también posee un método relacionado con los arrays, dicho método permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copia en el origen, el array destino de la copia, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia).

```
int uno[]={1,1,2};
int dos[]={3,3,3,3,3,3,3,3,3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0;i<=8;i++){
    System.out.print(dos[i]+" ");
} //Sale 112333333
```

clase String

introducción

Para Java las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo String. Ejemplo:

```
String texto1 = "¡Prueba de texto!";
```

Las cadenas pueden ocupar varias líneas utilizando el operador de concatenación "+".

```
String texto2 ="Este es un texto que ocupa " +
               "varias líneas, no obstante se puede "+
               "perfectamente encadenar";
```

También se pueden crear objetos String sin utilizar constantes entrecomilladas, usando otros constructores:

```
char[] palabra = {'P','a','l','b','r','a'};//Array de char
String cadena = new String(palabra);
byte[] datos = {97,98,99};
String codificada = new String (datos, "8859_1");
```

En el último ejemplo la cadena *codificada* se crea desde un array de tipo byte que contiene números que serán interpretados como códigos Unicode. Al asignar, el valor 8859_1 indica la tabla de códigos a utilizar.

comparación entre objetos String

Los objetos **String** no pueden compararse directamente con los operadores de comparación. En su lugar se deben utilizar estas expresiones:

- *cadena1*.**equals**(*cadena2*). El resultado es **true** si la cadena1 es igual a la cadena2. Ambas cadenas son variables de tipo **String.**
- *cadena1*.**equalsIgnoreCase**(cadena2). Como la anterior, pero en este caso no se tienen en cuenta mayúsculas y minúsculas.
- *s1*.**compareTo**(*s2*). Compara ambas cadenas, considerando el orden alfabético. Si la primera cadena es mayor en orden alfabético que la segunda devuelve 1, si son iguales devuelve 0 y si es la segunda la mayor devuelve -1. Hay que tener en cuenta que el orden no es el del alfabeto español, sino que usa la tabla ASCII, en esa tabla la letra *ñ* es mucho mayor que la *o*.
- *s1*.**compareToIgnoreCase**(*s2*). Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

String.valueOf

Este método pertenece no sólo a la clase String, sino a otras y siempre es un método que convierte valores de una clase a otra. En el caso de los objetos String, permite convertir valores que no son de cadena a forma de cadena. Ejemplos:

```
String numero = String.valueOf(1234);
String fecha = String.valueOf(new Date());
```

En el ejemplo se observa que este método pertenece a la clase String directamente, no hay que utilizar el nombre del objeto creado (como se verá más adelante, es un método estático).

métodos de las variables de las cadenas

Son métodos que poseen las propias variables de cadena. Para utilizarlos basta con poner el nombre del método y sus parámetros después del nombre de la variable String. Es decir: ***variableString.método(argumentos)***

length

Permite devolver la longitud de una cadena (el número de caracteres de la cadena):

```
String texto1="Prueba";
System.out.println(texto1.length());//Escribe 6
```

concatenar cadenas

Se puede hacer de dos formas, utilizando el método **concat** o con el operador +. Ejemplo:

```
String s1="Buenos ", s2="días", s3, s4;
s3 = s1 + s2;
s4 = s1.concat(s2);
```

charAt

Devuelve un carácter de la cadena. El carácter a devolver se indica por su posición (el primer carácter es la posición 0) Si la posición es negativa o sobrepasa el tamaño de la cadena, ocurre un error de ejecución, una excepción tipo **IndexOutOfBounds-Exception**. Ejemplo:

```
String s1="Prueba";
char c1=s1.charAt(2); //c1 valdrá 'u'
```

substring

Da como resultado una porción del texto de la cadena. La porción se toma desde una posición inicial hasta una posición final (sin incluir esa posición final). Si las posiciones indicadas no son válidas ocurre una excepción de tipo **IndexOutOfBounds-Exception.** Se empieza a contar desde la posición 0. Ejemplo:

```
String s1="Buenos días";
```

```
String s2=s1.substring(7,10); //s2 = día
```

indexOf

Devuelve la primera posición en la que aparece un determinado texto en la cadena. En el caso de que la cadena buscada no se encuentre, devuelve -1. El texto a buscar puede ser **char** o **String.** Ejemplo:

```
String  s1="Quería decirte que quiero que te vayas";
System.out.println(s1.indexOf("que")); //Da 15
```

Se puede buscar desde una determinada posición. En el ejemplo anterior:

```
System.out.println(s1.indexOf("que",16)); //Ahora da 26
```

lastIndexOf

Devuelve la última posición en la que aparece un determinado texto en la cadena. Es casi idéntica a la anterior, sólo que busca desde el final. Ejemplo:

```
String  s1="Quería decirte que quiero que te vayas";
System.out.println(s1.lastIndexOf("que"); //Da 26
```

También permite comenzar a buscar desde una determinada posición.

endsWith

Devuelve **true** si la cadena termina con un determinado texto. Ejemplo:

```
String  s1="Quería decirte que quiero que te vayas";
System.out.println(s1.endsWith("vayas"); //Da true
```

startsWith

Devuelve **true** si la cadena empieza con un determinado texto.

replace

Cambia todas las apariciones de un carácter por otro en el texto que se indique y lo almacena como resultado. El texto original no se cambia, por lo que hay que asignar el resultado de **replace** a un String para almacenar el texto cambiado:

```
String s1="Mariposa";
System.out.println(s1.replace('a','e'));//Da Meripose
System.out.println(s1);//Sigue valiendo Mariposa
```

replaceAll

Modifica en un texto cada entrada de una cadena por otra y devuelve el resultado. El primer parámetro es el texto que se busca (que puede ser una expresión regular), el segundo parámetro es el texto con el que se reemplaza el buscado. La cadena original no se modifica.

```
String s1="Cazar armadillos";
System.out.println(s1.replace("ar","er"));//Da Cazer ermedillos
System.out.println(s1);//Sigue valiendo Cazar armadilos
```

toUpperCase

Devuelve la versión en mayúsculas de la cadena.

toLowerCase

Devuelve la versión en minúsculas de la cadena.

toCharArray

Obtiene un array de caracteres a partir de una cadena.

lista completa de métodos

método	descripción
char charAt(int index**)**	Proporciona el carácter que está en la posición dada por el entero *index*.
int compareTo(string s**)**	Compara las dos cadenas. Devuelve un valor menor que cero si la cadena *s* es mayor que la original, devuelve 0 si son iguales y devuelve un valor mayor que cero si *s* es menor que la original.
int compareToIgnoreCase(string s**)**	Compara dos cadenas, pero no tiene e cuenta si el texto es mayúsculas o no.
String concat(String s**)**	Añade la cadena *s* a la cadena original.
String copyValueOf(char[] data**)**	Produce un objeto **String** que es igual al array de caracteres *data*.
boolean endsWith(String s**)**	Devuelve **true** si la cadena termina con el texto *s*
boolean equals(String s**)**	Compara ambas cadenas, devuelve **true** si son iguales
boolean equalsIgnoreCase(String s**)**	Compara ambas cadenas sin tener en cuenta las mayúsculas y las minúsculas.
byte[] getBytes()	Devuelve un array de caracteres que toma a partir de la cadena de texto
void getBytes(int srcBegin, **int** srcEnd, **char[]** dest, **int** dstBegin);	Almacena el contenido de la cadena en el array de caracteres *dest*. Toma los caracteres desde la posición *srcBegin* hasta la posición *srcEnd* y les copia en el array desde la posición *dstBegin*
int indexOf(String s**)**	Devuelve la posición en la cadena del texto s
int indexOf(String s, **int** primeraPos)	Devuelve la posición en la cadena del texto s, empezando a buscar desde la posición *PrimeraPos*
int lastIndexOf(String s**)**	Devuelve la última posición en la cadena del texto s

método	descripción
int **lastIndexOf(**String s, int primeraPos)	Devuelve la última posición en la cadena del texto s, empezando a buscar desde la posición *PrimeraPos*
int length()	Devuelve la longitud de la cadena
String **replace(**char carAnterior, char ncarNuevo)	Devuelve una cadena idéntica al original pero que ha cambiando los caracteres iguales a *carAnterior* por *carNuevo*
String **replaceFirst(**String str1, String str2**)**	Cambia la primera aparición de la cadena str1 por la cadena str2
String **replaceFirst(**String str1, String str2**)**	Cambia la primera aparición de la cadena uno por la cadena dos
String **replaceAll(**String str1, String str2**)**	Cambia la todas las apariciones de la cadena uno por la cadena dos
String **startsWith(**String s**)**	Devuelve **true** si la cadena comienza con el texto *s*.
String **substring(**int primeraPos, int segundaPos**)**	Devuelve el texto que va desde *primeraPos* a *segunaPos*.
char[] **toCharArray()**	Devuelve un array de caracteres a partir de la cadena dada
String **toLowerCase()**	Convierte la cadena a minúsculas
String **toLowerCase(**Locale local**)**	Lo mismo pero siguiendo las instrucciones del argumento *local*
String **toUpperCase()**	Convierte la cadena a mayúsculas
String **toUpperCase(**Locale local**)**	Lo mismo pero siguiendo las instrucciones del argumento *local*
String **trim()**	Elimina los blancos que tenga la cadena tanto por delante como por detrás
Static String **valueOf(***tipo* elemento**)**	Devuelve la cadena que representa el valor *elemento*. Si elemento es booleano, por ejemplo devolvería una cadena con el valor **true** o **false**

objetos y clases

programación orientada a objetos

Se ha comentado anteriormente en este manual que Java es un lenguaje totalmente orientado a objetos. De hecho siempre hemos definido una clase pública con un método **main** que permite que se pueda visualizar en la pantalla el programa Java.

La gracia de la POO es que se hace que los problemas sean más sencillos, al permitir dividir el problema. Está división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente. Un objeto es un elemento del programa que posee sus propios datos y su propio funcionamiento.

Es decir un objeto está formado por datos (**propiedades**) y funciones que es capaz de realizar el objeto (**métodos**).

Antes de poder utilizar un objeto, se debe definir su **clase.** La clase es la definición de un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase **ficha** definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.

propiedades de la POO

- **Encapsulamiento**. Una clase se compone tanto de variables (propiedades) como de funciones y procedimientos (métodos). De hecho no se pueden definir variables (ni funciones) fuera de una clase (es decir no hay variables *globales*).

- **Ocultación**. Hay una zona oculta al definir la clases (zona privada) que sólo es utilizada por esa clases y por alguna clase relacionada. Hay una zona pública (llamada también **interfaz** de la clase) que puede ser utilizada por cualquier parte del código.

- **Polimorfismo**. Cada método de una clase puede tener varias definiciones distintas. En el caso del parchís: partida.empezar(4) empieza una partida para cuatro jugadores, partida.empezar(rojo, azul) empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método empezar, que es polimórfico.

- **Herencia**. Una clase puede heredar propiedades de otra.

introducción al concepto de objeto

Un objeto es cualquier entidad representable en un programa informático, bien sea real (ordenador) o bien sea un concepto (transferencia). Un objeto en un sistema posee: una identidad, un estado y un comportamiento.

El **estado** marca las condiciones de existencia del objeto dentro del programa. Lógicamente este estado puede cambiar. Un coche puede estar parado, en marcha, estropeado, funcionando, sin gasolina, etc.

El **comportamiento** determina como responde el objeto ante peticiones de otros objetos. Por ejemplo un objeto conductor puede lanzar el mensaje arrancar a un coche. El comportamiento determina qué es lo que hará el objeto.

La **identidad** determina que cada objeto es único aunque tengan el mismo valor. No existen dos objetos iguales. Lo que sí existe es dos referencias al mismo objeto.

Los objetos se manejan por referencias, existirá una referencia a un objeto. De modo que esa referencia permitirá cambiar los atributos del objeto. Incluso puede haber varias referencias al mismo objeto, de modo que si una referencia cambia el estado del objeto, el resto (lógicamente) mostrarán esos cambios.

Los objetos por valor son los que no usan referencias y usan copias de valores concretos. En Java estos objetos son los tipos simples: **int, char, byte, short, long, float, double** y **boolean.** El resto son todos objetos (incluidos los arrays y Strings).

clases

Las clases son las plantillas para hacer objetos. Una clase sirve para definir una serie de objetos con propiedades (atributos), comportamientos (operaciones o métodos), y semántica comunes. Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- **Sus atributos.** Es decir, los datos miembros de esa clase. Los datos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase. También se las llama campos.
- **Sus métodos.** Las funciones miembro de la clase. Son las acciones (u operaciones) que puede realizar la clase.
- **Código de inicialización**. Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el constructor de la clase).
- **Otras clases**. Dentro de una clase se pueden definir otras clases (clases internas, son consideradas como asociaciones dentro de UML).

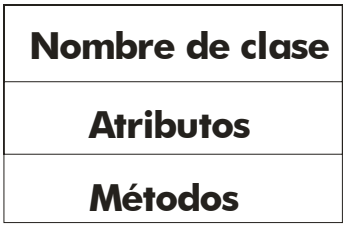


Ilustración 5, Clase en notación UML

El formato general para crear una clase en Java es:

```
[acceso] class nombreDeClase {
   [acceso] [static] tipo atributo1;
   [acceso] [static] tipo atributo2;
   [acceso] [static] tipo atributo3;
   ...
   [access] [static] tipo método1(listaDeArgumentos) {
         ...código del método...
   }
   ...
}
```

La palabra opcional **static** sirve para hacer que el método o la propiedad a la que precede se pueda utilizar de manera genérica (más adelante se hablará de clases genéricas), los métodos o propiedades así definidos se llaman **atributos de clase** y **métodos de clase** respectivamente. Su uso se verá más adelante. Ejemplo;

```
class Noria {
   double radio;
   void girar(int velocidad){
         ...//definición del método
   }
   void parar(){...
}
```

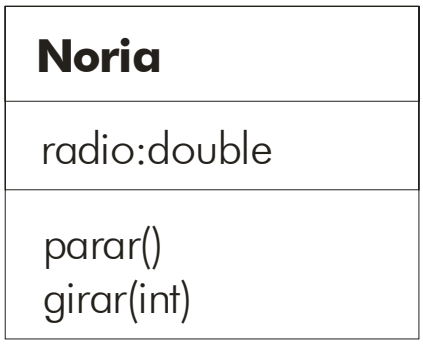


Ilustración 6, Clase Noria bajo notación UML

objetos

Se les llama **instancias de clase**. Son un elemento en sí de la clase (en el ejemplo del parchís, una ficha en concreto). Un objeto se crea utilizando el llamado **constructor** de la clase. El constructor es el método que permite iniciar el objeto.

datos miembro (propiedades o atributos)

Para poder acceder a los atributos de un objeto, se utiliza esta sintaxis:

```
objeto.atributo
```

Por ejemplo:

```
Noria.radio;
```

métodos

Los métodos se utilizan de la misma forma que los atributos, excepto porque los métodos poseen siempre paréntesis, dentro de los cuales pueden ir valore snecesarios para la ejecución del método (parámetros):

```
objeto.método(argumentosDelMétodo)
```

Los métodos siempre tienen paréntesis (es la diferencia con las propiedades) y dentro de los paréntesis se colocan los argumentos del método. Que son los datos que necesita el método para funcionar. Por ejemplo:

```
MiNoria.gira(5);
```

Lo cual podría hacer que la Noria avance a 5 Km/h.

herencia

En la POO tiene mucha importancia este concepto, la herencia es el mecanismo que permite crear clases basadas en otras existentes. Se dice que esas clases *descienden* de las primeras. Así por ejemplo, se podría crear una clase llamada **vehículo** cuyos métodos serían mover*, parar, acelerar* y *frenar*. Y después se podría crear una clase **coche** basada en la anterior que tendría esos mismos métodos (les heredaría) y además añadiría algunos propios, por ejemplo *abrirCapó* o *cambiarRueda*.

creación de objetos de la clase

Una vez definida la clase, se pueden utilizar objetos de la clase. Normalmente consta de dos pasos. Su declaración, y su creación. La declaración consiste en indicar que se va a utilizar un objeto de una clase determinada. Y se hace igual que cuando se declara una variable simple. Por ejemplo:

```
Noria noriaDePalencia;
```

Eso declara el objeto *noriaDePalencia* como objeto de tipo *Noria;* se supone que previamente se ha definido la clase *Noria*.

Para poder utilizar un objeto, hay que crearle de verdad. Eso consiste en utilizar el operador **new.** Por ejemplo:

```
noriaDePalencia = new Noria();
```

Al hacer esta operación el objeto reserva la memoria que necesita y se inicializa el objeto mediante su **constructor**. Más adelante veremos como definir el constructor.

Ilustración 7, *Objeto NoriaDePalencia de la clase Noria en notación UML*

especificadores de acceso

Se trata de una palabra que antecede a la declaración de una clase, método o propiedad de clase. Hay tres posibilidades: **public, protected** y **private.** Una cuarta posibilidad es no utilizar ninguna de estas tres palabras; entonces se dice que se ha utilizado el modificador por defecto (***friendly***).

Los especificadores determinan el alcance de la visibilidad del elemento al que se refieren. Referidos por ejemplo a un método, pueden hacer que el método sea visible sólo para la clase que lo utiliza (**private**), para éstas y las heredadas (**protected**), para todas las clases del mismo paquete (***friendly***) o para cualquier clase del tipo que sea (**public**).

En la siguiente tabla se puede observar la visibilidad de cada especificador:

zona	*private* (privado)	sin modificador *(friendly)*	*protected* (protegido)	*public* (público)
Misma clase	**X**	**X**	**X**	**X**
Subclase en el mismo paquete		**X**	**X**	**X**
Clase (no subclase) en el mismo paquete		**X**		**X**
Subclase en otro paquete			**X**	**X**
No subclase en otro paquete				**X**

creación de clases

definir atributos de la clase (variables, propiedades o datos de la clases)

Cuando se definen los datos de una determinada clase, se debe indicar el tipo de propiedad que es (String, int, double, int[][],...) y el **especificador de acceso** (public, private,...). El especificador indica en qué partes del código ese dato será visible.

Ejemplo:

```
class Persona {
  public String nombre;//Se puede acceder desde cualquier clase
  private int contraseña;//Sólo se puede acceder desde la
                         //clase Persona
  protected String dirección; //Acceden a esta propiedad
                        //esta clase y sus descendientes
```

Por lo general las propiedades de una clase suelen ser privadas o protegidas, a no ser que se trate de un valor constante, en cuyo caso se declararán como públicos.

Las variables locales de una clase pueden ser inicializadas.

```
class auto{
  public nRuedas=4;
```

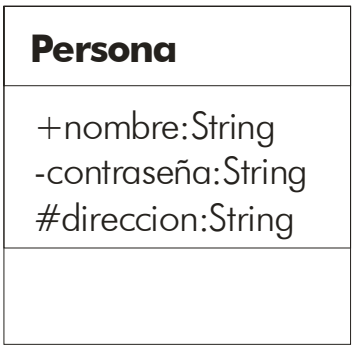


Ilustración 8, La clase persona en UML. El signo + significa *public*, el signo # *protected* y el signo - *private*

definir métodos de clase (operaciones o funciones de clase)

Un método es una llamada a una operación de un determinado objeto. Al realizar esta llamada (también se le llama enviar un mensaje), el control del programa pasa a ese método y lo mantendrá hasta que el método finalice o se haga uso de **return**.

Para que un método pueda trabajar, normalmente hay que pasarle unos datos en forma de argumentos o parámetros, cada uno de los cuales se separa por comas. Ejemplos de llamadas:

```
balón.botar(); //sin argumentos
miCoche.acelerar(10);
```

```
ficha.comer(posición15);posición 15 es una variable que se
             //pasa como argumento
partida.empezarPartida("18:15",colores);
```

Los métodos de la clase se definen dentro de ésta. Hay que indicar un modificador de acceso (**public, private, protected** o ninguno, al igual que ocurre con las variables y con la propia clase) y un tipo de datos, que indica qué tipo de valores devuelve el método.

Esto último se debe a que los métodos son funciones que pueden devolver un determinado valor (un entero, un texto, un valor lógico,...) mediante el comando **return**. Si el método no devuelve ningún valor, entonces se utiliza el tipo **void** que significa que no devuelve valores (en ese caso el método no tendrá instrucción **return**).

El último detalle a tener en cuenta es que los métodos casi siempre necesitan datos para realizar la operación, estos datos van entre paréntesis y se les llama argumentos. Al definir el método hay que indicar que argumentos se necesitan y de qué tipo son.

Ejemplo:

```
public class vehiculo {
    /** Función principal */
    int ruedas;
    private double velocidad=0;
    String nombre;
    /** Aumenta la velocidad*/
     public void acelerar(double cantidad) {
        velocidad += cantidad;
    }
    /** Disminuye la velocidad*/
     public void  frenar(double cantidad) {
        velocidad -= cantidad;
    }
    /** Devuelve la velocidad*/
    public double obtenerVelocidad(){
         return velocidad;
     }

public static void main(String args[]){
      vehiculo miCoche = new vehiculo();
      miCoche.acelerar(12);
      miCoche.frenar(5);
      System.out.println(miCoche.obtenerVelocidad());
    } // Da 7.0
```

En la clase anterior, los métodos **acelerar** y **frenar** son de tipo **void** por eso no tienen sentencia **return**. Sin embargo el método **obtenerVelocidad** es de tipo double por lo que su resultado es devuelto por la sentencia **return** y puede ser escrito en pantalla.

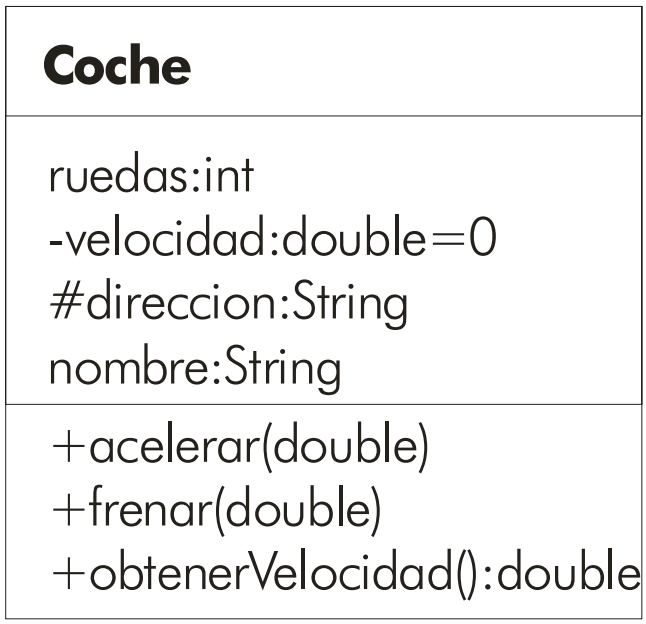


Ilustración 9, Versión UML de la clase Coche

argumentos por valor y por referencia

En todos los lenguajes éste es un tema muy importante. Los argumentos son los datos que recibe un método y que necesita para funcionar. Ejemplo:

```
public class Matemáticas {
  public double factorial(int n){
        double resultado;
        for (resultado=n;n>1;n--) resultado*=n;
        return resultado;
  }
  ...
public static void main(String args[]){
  Matemáticas m1=new Matemáticas();
  double x=m1.factorial(25);//Llamada al método
  }
```

En el ejemplo anterior, el valor 25 es un argumento requerido por el método **factorial** para que éste devuelva el resultado (que será el factorial de 25). En el código del método factorial, este valor 25 es copiado a la variable **n,** que es la encargada de almacenar y utilizar este valor.

Se dice que los argumentos son por valor, si la función recibe una copia de esos datos, es decir la variable que se pasa como argumento no estará afectada por el código. Ejemplo:

```
class prueba {
  public void metodo1(int entero){
        entero=18;
...
  }
...
public static void main(String args[]){
        int x=24;
       prueba miPrueba = new prueba();
        miPrueba.metodo1(x);
        System.out.println(x); //Escribe 24, no 18
}
```

Este es un ejemplo de paso de parámetros por valor. La variable x se pasa como argumento o parámetro para el método *metodo1*, allí la variable *entero* recibe una **copia** del **valor** de x en la variable **entero**, y a esa copia se le asigna el valor 18. Sin embargo la variable x no está afectada por esta asignación.

Sin embargo en este otro caso:

```
class prueba {
  public void metodo1(int[] entero){
        entero[0]=18;
...
  }
...
public static void main(String args[]){
        int x[]={24,24};
       prueba miPrueba = new prueba();
        miPrueba.metodo1(x);
        System.out.println(x[0]); //Escribe 18, no 24
```

Aquí sí que la variable x está afectada por la asignación entero[0]=18. La razón es porque en este caso el método no recibe el valor de esta variable, sino la **referencia**, es decir la dirección física de esta variable. *entero* no es una replica de *x*, es la propia x llamada de otra forma.

Los tipos básicos (**int, double, char, boolean, float, short** y **byte**) se pasan por valor. También se pasan por valor las variables **String.** Los objetos y arrays se pasan por referencia.

devolución de valores

Los métodos pueden devolver valores básicos (int, short, double, etc.), Strings, arrays e incluso objetos.

En todos los casos es el comando **return** el que realiza esta labor. En el caso de arrays y objetos, devuelve una referencia a ese array u objeto. Ejemplo:

```
class FabricaArrays {
    public int[] obtenArray(){
        int array[]= {1,2,3,4,5};
        return array;
    }
}

public class returnArray {
    public static void main(String[] args) {
        FabricaArrays fab=new FabricaArrays();
        int nuevoArray[]=fab.obtenArray();
    }
}
```

sobrecarga de métodos

Una propiedad de la POO es el polimorfismo. Java posee esa propiedad ya que admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método. Ejemplo:

```
class Matemáticas{
  public double suma(double x, double y) {
        return x+y;
  }
  public double suma(double x, double y, double z){
        return x+y+z;
  }
  public double suma(double[] array){
        double total =0;
        for(int i=0; i<array.length;i++){
              total+=array[i];
        }
        return total;
  }
```

La clase matemáticas posee tres versiones del método suma. una versión que suma dos números double, otra que suma tres y la última que suma todos los miembros de un array de doubles. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

la referencia *this*

La palabra **this** es una referencia al propio objeto en el que estamos. Ejemplo:

```
class punto {
  int posX, posY;//posición del punto
  punto(posX, posY){
        this.posX=posX;
        this.posY=posY;
  }
```

En el ejemplo hace falta la referencia **this** para clarificar cuando se usan las propiedades *posX y posY,* y cuando los argumentos con el mismo nombre. Otro ejemplo:

```
class punto {
  int posX, posY;
  ...
        /**Suma las coordenadas de otro punto*/
        public void suma(punto punto2){
              posX = punto2.posX;
              posY = punto2.posY;
        }

  /** Dobla el valor de las coordenadas del punto*/
        public void dobla(){
              suma(this);
        }
```

En el ejemplo anterior, la función dobla, dobla el valor de las coordenadas pasando el propio punto como referencia para la función suma (un punto sumado a sí mismo, daría el doble).

Los posibles usos de **this** son:

- **this.** Referencia al objeto actual. Se usa por ejemplo pasarle como parámetro a un método cuando es llamado desde la propia clase.
- **this.***atributo*. Para acceder a una propiedad del objeto actual.
- **this.***método***(***parámetros***).** Permite llamar a un método del objeto actual con los parámetros indicados.
- **this(***parámetros***).** Permite llamar a un constructor del objeto actual. Esta llamada sólo puede ser empleada en la primera línea de un constructor.

creación de constructores

Un constructor es un método que es llamado automáticamente al crear un objeto de una clase, es decir al usar la instrucción **new**. Sin embargo en ninguno de los ejemplos anteriores se ha definido constructor alguno, por eso no se ha utilizado ningún constructor al crear el objeto.

Un constructor no es más que un método que tiene el mismo nombre que la clase. Con lo cual para crear un constructor basta definir un método en el código de la clase que tenga el mismo nombre que la clase. Ejemplo:

```
class Ficha {
  private int casilla;
  Ficha() { //constructor
        casilla = 1;
  }
  public void avanzar(int n) {
        casilla += n;
  }
  public int casillaActual(){
        return casilla;
  }
}

public class app {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha();
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 4
```

En la línea *Ficha ficha1 = new Ficha();* es cuando se llama al constructor, que es el que coloca inicialmente la casilla a 1. Pero el constructor puede tener parámetros:

```
class Ficha {
  private int casilla; //Valor inicial de la propiedad
  Ficha(int n) { //constructor
        casilla = n;
  }
  public void avanzar(int n) {
        casilla += n;
  }
  public int casillaActual(){
        return casilla;
  }
}
```

```
public class app {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha(6);
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 9
```

En este otro ejemplo, al crear el objeto *ficha1*, se le da un valor a la casilla, por lo que la casilla vale al principio **6**.

Hay que tener en cuenta que puede haber más de un constructor para la misma clase. Al igual que ocurría con los métodos, los constructores se pueden sobrecargar.

De este modo en el código anterior de la clase *Ficha* se podrían haber colocado los dos constructores que hemos visto, y sería entonces posible este código:

```
Ficha ficha1= new Ficha(); //La propiedad casilla de la
                           //ficha valdrá 1
Ficha ficha1= new Ficha(6); //La propiedad casilla de la
                            //ficha valdrá 6
```

Cuando se sobrecargan los constructores (se utilizan varias posibilidades de constructor), se pueden hacer llamadas a constructores mediante el objeto **this**

métodos y propiedades genéricos (*static*)

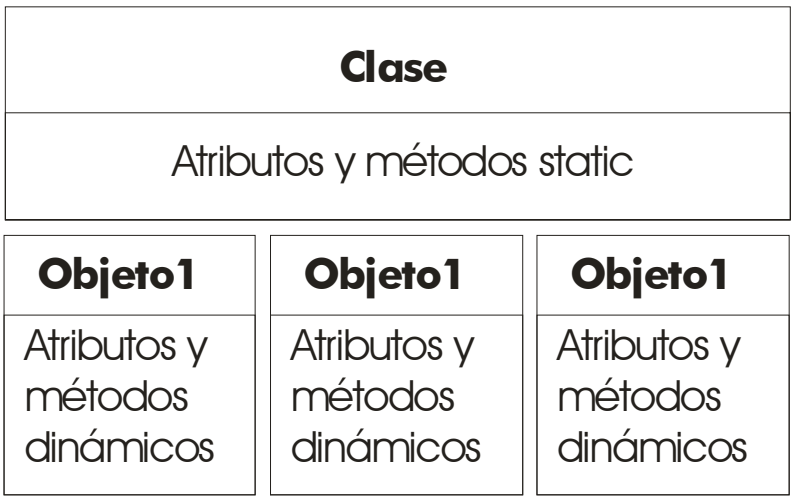


Ilustración 10, *Diagrama de funcionamiento de los métodos y atributos static*

Hemos visto que hay que crear objetos para poder utilizar los métodos y propiedades de una determinada clase. Sin embargo esto no es necesario si la propiedad o el método se definen precedidos de la palabra clave **static**. De esta forma se podrá utilizar el método sin definir objeto alguno, utilizando el nombre de la clase como si fuera un objeto. Así funciona la clase **Math** (véase la clase Math, página 23). Ejemplo:

```
class Calculadora {
   static public int factorial(int n) {
         int fact=1;
         while (n>0) {
               fact *=n--;
         }
         return fact;
   }
}
public class app {
   public static void main(String[] args) {
         System.out.println(Calculadora.factorial(5));
   }
}
```

En este ejemplo no ha hecho falta crear objeto alguno para poder calcular el factorial.

Una clase puede tener métodos y propiedades genéricos (static) y métodos y propiedades dinámicas (normales).

Cada vez que se crea un objeto con **new**, se almacena éste en memoria. Los métodos y propiedades normales, gastan memoria por cada objeto que se cree, sin embargo los métodos estáticos no gastan memoria por cada objeto creado, gastan memoria al definir la clase sólo. Es decir los métodos y atributos static son los mismos para todos los objetos creados, gastan por definir la clase, pero no por crear cada objeto.

Hay que crear métodos y propiedades genéricos cuando ese método o propiedad vale o da el mismo resultado en todos los objetos. Pero hay que utilizar métodos normales (dinámicos) cuando el método da resultados distintos según el objeto. Por ejemplo en un clase que represente aviones, la altura sería un atributo dinámico (distinto en cada objeto), mientras que el número total de aviones, sería un método static (es el mismo para todos los aviones).

el método main

Hasta ahora hemos utilizado el método main de forma incoherente como único posible mecanismo para ejecutar programas. De hecho este método dentro de una clase, indica que la clase es ejecutable desde la consola. Su prototipo es:

```
public static void main(String[] args){
   ...instruccionesejecutables....
}
```

Hay que tener en cuenta que el método main es estático, por lo que no podrá utilizar atributos o métodos dinámicos de la clase.

Los argumentos del método main son un array de caracteres donde cada elemento del array es un parámetro enviado por el usuario desde la línea de comandos. A este argumento se le llama comúnmente *args*. Es decir, si se ejecuta el programa con:

```
java claseConMain uno dos
```

Entonces el método main de esta clase recibe un array con dos elementos, el primero es la cadena "uno" y el segundo la cadena "dos" (es decir args[0]="uno"; args[1]="dos").

destrucción de objetos

En C y C++ todos los programadores saben que los objetos se crean con **new** y para eliminarles de la memoria y así ahorrarla, se deben eliminar con la instrucción **delete.** Es decir, es responsabilidad del programador eliminar la memoria que gastaban los objetos que se van a dejar de usar. La instrucción **delete** del C++ llama al destructor de la clase, que es una función que se encarga de eliminar adecuadamente el objeto.

La sorpresa de los programadores C++ que empiezan a trabajar en Java es que **no hay instrucción *delete* en Java.** La duda está entonces, en cuándo se elimina la memoria que ocupa un objeto.

En Java hay un recolector de basura (***garbage collector***) que se encarga de gestionar los objetos que se dejan de usar y de eliminarles de memoria. Este proceso es automático e impredecible y trabajo en un hilo (***thread***) de baja prioridad.

Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un programa. Esta eliminación depende de la máquina virtual, en casi todas la recolección se realiza periódicamente en un determinado lapso de tiempo. La implantación de máquina virtual conocida como HotSpot[1] suele hacer la recolección mucho más a menudo

Se puede forzar la eliminación de un objeto asignándole el valor **null**, pero teniendo en cuenta que eso no equivale al famoso **delete** del lenguaje C++. Con null no se libera inmediatamente la memoria, sino que pasará un cierto tiempo (impredecible, por otro lado) hasta su total destrucción.

Se puede invocar al recolector de basura desde el código invocando al método estático `System.gc()`. Esto hace que el recolector de basura trabaje en cuanto se lea esa invocación.

Sin embargo puede haber problemas al crear referencias circulares. Como:

```
class uno {
  dos d;
  uno() { //constructor
        d = new dos();
  }
}

class dos {
  uno u;
  dos() {
        u = new uno();
  }

public class app {
```

[1] Para saber más sobre HotSpot acudir a java.sun.com/products/hotspot/index.html.

```
  public static void main(Stgring[] args)  {
        uno prueba = new uno();//referencia circular
        prueba = null; //no se liberará bien la memoria
  }
}
```

Al crear un objeto de clase uno, automáticamente se crea uno de la clase dos, que al crearse creará otro de la clase uno. Eso es un error que provocará que no se libere bien la memoria salvo que se eliminen previamente los objetos referenciados.

el método *finalize*

Es equivalente a los destructores del C++. Es un método que es llamado antes de eliminar definitivamente al objeto para hacer limpieza final. Un uso puede ser eliminar los objetos creados en la clase para eliminar referencias circulares. Ejemplo:

```
class uno {
  dos d;
  uno() {
        d = new dos();
  }
  protected void finalize(){
        d = null;//Se elimina d por lo que pudiera pasar
  }
}
```

finalize es un método de tipo **protected** heredado por todas las clases ya que está definido en la clase raíz **Object.**

La diferencia de finalize respecto a los métodos destructores de C++ estriba en que en Java no se llaman al instante (de hecho es imposible saber cuando son llamados). la llamada **System.gc()** llama a todos los finalize pendientes inmediatamente (es una forma de probar si el método finalize funciona o no).

reutilización de clases

herencia

introducción

Es una de las armas fundamentales de la programación orientada a objetos. Permite crear nuevas clases que heredan características presentas en clases anteriores. Esto facilita enormemente el trabajo porque ha permitido crear clases estándar para todos los programadores y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear nuestras clases desde cero.

Para que una clase herede las características de otra hay que utilizar la palabra clave **extends** tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. Sólo se puede tener herencia de una clase (a la clase de la que se hereda se la llama **superclase** y a la clase heredada se la llama **subclase**). Ejemplo:

```
class coche extends vehiculo {
...
} //La clase coche parte de la definición de vehículo
```

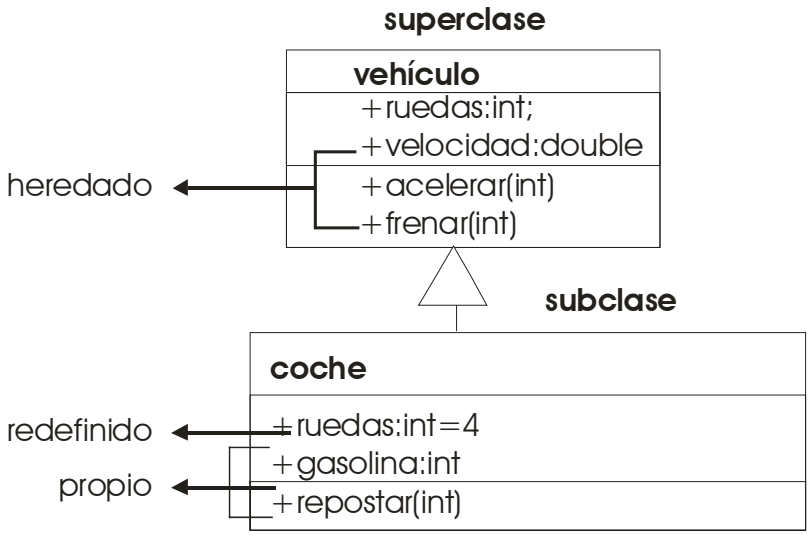


Ilustración 11, herencia

métodos y propiedades no heredados

Por defecto se heredan todos los métodos y propiedades *protected* y *public* (no se heredan los *private*). Además si se define un método o propiedad en la subclase con el mismo nombre que en la superclase, entonces se dice que se está redefiniendo el método, con lo cual no se hereda éste, sino que se reemplaza por el nuevo.

Ejemplo:

```
class vehiculo {
  public int velocidad;
  public int ruedas;
  public void parar() {
        velocidad = 0;
  }
  public void acelerar(int kmh) {
        velocidad += kmh;
  }

class coche extends vehiculo{
  public int ruedas=4;
  public int gasolina;
  public void repostar(int litros) {
        gasolina+=litros;
  }
}
.....................................................
public class app {
  public static void main(String[] args) {
        coche coche1=new coche();
        coche.acelerar(80);//Método heredado
        coche.repostar(12);
```

anulación de métodos

Como se ha visto, las subclases heredan los métodos de las superclases. Pero es más, también los pueden sobrecargar para proporcionar una versión de un determinado método.

Por último, si una subclase define un método con el mismo nombre, tipo y argumentos que un método de la superclase, se dice entonces que se sobrescribe o anula el método de la superclase. Ejemplo:

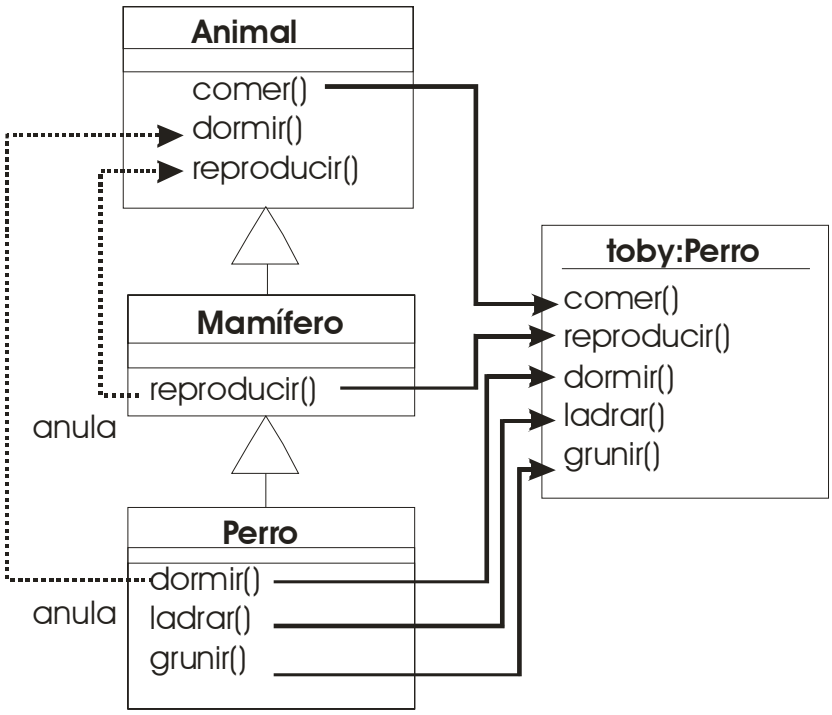


Ilustración 12, anulación de métodos

super

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada **super**. Si **this** hace referencia a la clase actual, **super** hace referencia a la superclase respecto a la clase actual, con lo que es un método imprescindible para poder acceder a métodos anulados por herencia. Ejemplo

```
public class vehiculo{
   double velocidad;
   ...
   public void acelerar(double cantidad){
         velocidad+=cantidad;
   }
}

public class coche extends vehiculo{
   double gasolina;
   public void acelerar(double cantidad){
         super.acelerar(cantidad);
         gasolina*=0.9;
   }
```

En el ejemplo anterior, la llamada *super.acelerar(cantidad)* llama al método acelerar de la clase vehículo (el cual acelerará la marcha). Es necesario redefinir el método acelerar

en la clase coche ya que aunque la velocidad varía igual que en la superclase, hay que tener en cuenta el consumo de gasolina

Se puede incluso llamar a un **constructor** de una superclase, usando la sentencia **super()**. Ejemplo:

```
public class vehiculo{
   double velocidad;
   public vehiculo(double v){
         velocidad=v;
   }
}

public class coche extends vehiculo{
   double gasolina;
   public coche(double v, double g){
         super(v);//Llama al constructor de la clase vehiculo
         gasolina=g
   }
}
```

Por defecto Java realiza estas acciones:

- Si la primera instrucción de un constructor de una subclase es una sentencia que no es ni **super** ni **this**, Java añade de forma invisible e implícita una llamada **super()** al constructor por defecto de la superclase, luego inicia las variables de la subclase y luego sigue con la ejecución normal.
- Si se usa **super(..)** en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.
- Finalmente, si esa primera instrucción es **this(..)**, entonces se llama al constructor seleccionado por medio de **this**, y después continúa con las sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante **this**.

casting de clases

Como ocurre con los tipos básicos (ver conversión entre tipos *(casting)*, página 18, es posible realizar un casting de objetos para convertir entre clases distintas. Lo que ocurre es que sólo se puede realizar este *casting* entre subclases. Es decir se realiza un casting para especificar más una referencia de clase (se realiza sobre una superclase para convertirla a una referencia de una subclase suya).

En cualquier otro caso no se puede asignar un objeto de un determinado tipo a otro.

Ejemplo:

```
Vehiculo vehiculo5=new Vehiculo();
Coche cocheDePepe = new Coche("BMW");
vehiculo5=cocheDePepe //Esto sí se permite
cocheDePepe=vehiculo5;//Tipos incompatibles
cocheDepepe=(coche)vehiculo5;//Ahora sí se permite
```

Hay que tener en cuenta que los objetos nunca cambian de tipo, se les prepara para su asignación pero no pueden acceder a propiedades o métodos que no les sean propios. Por ejemplo, si *repostar()* es un método de la clase *coche* y no de *vehículo*:

```
Vehiculo v1=new Vehiculo();
Coche c=new Coche();
v1=c;//No hace falta casting
v1.repostar(5);//¡¡¡Error!!!
```

Cuando se fuerza a realizar un casting entre objetos, en caso de que no se pueda realizar ocurrirá una excepción del tipo **ClassCastingException.** Realmente sólo se puede hacer un *casting* si el objeto originalmente era de ese tipo. Es decir la instrucción:

```
cocheDepepe=(Coche) vehiculo4;
```

Sólo es posible si vehiculo4 hace referencia a un objeto coche.

instanceof

Permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

```
objeto instanceof clase
```

Comprueba si el objeto pertenece a una determinada clase y devuelve un valor **true** si es así. Ejemplo:

```
Coche miMercedes=new Coche();
if (miMercedes instanceof Coche)
  System.out.println("ES un coche");
if (miMercedes instanceof Vehículo)
  System.out.println("ES un coche");
if (miMercedes instanceof Camión)
  System.out.println("ES un camión");
```

En el ejemplo anterior aparecerá en pantalla:

```
ES un coche
ES un vehiculo
```

clases abstractas

A veces resulta que en las superclases se desean incluir métodos teóricos, métodos que no se desea implementar del todo, sino que sencillamente se indican en la clase para que el desarrollador que desee crear una subclase heredada de la clase abstracta, esté obligado a sobrescribir el método.

A las clases que poseen métodos de este tipo (métodos abstractos) se las llama **clases abstractas.** Son clases creadas para ser heredadas por nuevas clases creadas por el programador. Son clases base para herencia. Las clases abstractas no deben de ser instanciadas (no se pueden crear objetos de las clases abstractas).

Una clase abstracta debe ser marcada con la palabra clave **abstract.** Cada método abstracto de la clase, también llevará el *abstract*. Ejemplo:

```
abstract class vehiculo {
  public int velocidad=0;
  abstract public void acelera();
  public void para() {velocidad=0;}
}

class coche extends vehiculo {
  public void acelera() {
        velocidad+=5;
  }
}

public class prueba {
  public static void main(String[] args) {
        coche c1=new coche();
        c1.acelera();
        System.out.println(c1.velocidad);
        c1.para();
        System.out.println(c1.velocidad);
  }
}
```

final

Se trata de una palabra que se coloca antecediendo a un método, variable o clase. Delante de un método en la definición de clase sirve para indicar que ese método no puede ser sobrescrito por las subclases. Si una subclase intentar sobrescribir el método, el compilador de Java avisará del error.

Si esa misma palabra se coloca delante de una clase, significará que esa clase no puede tener descendencia.

Por último si se usa la palabra **final** delante de la definición de una propiedad de clase, entonces esa propiedad pasará a ser una constante, es decir no se le podrá cambiar el valor en ninguna parte del código.

clases internas

Se llaman clases internas a las clases que se definen dentro de otra clase. Esto permite simplificar aun más el problema de crear programas. Ya que un objeto complejo se puede descomponer en clases más sencillas. Pero requiere esta técnica una mayor pericia por parte del programador.

Al definir una clase dentro de otra, estamos haciéndola totalmente dependiente. Normalmente se realiza esta práctica para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.

Por ejemplo:

```
public class Coche {
    public int velocidad;
    public Motor motor;

    public Coche(int cil) {
        motor=new Motor(cil);
        velocidad=0;
    }

    public class Motor{ //Clase interna
        public int cilindrada;
        public Motor(int cil){
            cilindrada=cil;
        }
    }
}
```

El objeto *motor* es un objeto de la clase *Motor* que es interna a *Coche*. Si quisiéramos acceder al objeto motor de un coche sería:

```
Coche c=new Coche(1200);
System.out.println(c.motor.cilindrada);//Saldrá 1200
```

Las clases internas pueden ser privadas, protegidas o públicas. Fuera de la clase contenedora no pueden crear objetos (sólo se pueden crear *motores* dentro de un *coche*), salvo que la clase interna sea **static** en ese caso sí podrían. Por ejemplo (si la clase motor fuera estática):

```
//suponiendo que la declaración del Motor dentro de Coche es
// public class static Motor{....
Coche.Motor m=new Coche.Motor(1200);
```

Pero eso sólo tiene sentido si todos los Coches tuvieran el mismo motor.

Dejando de lado el tema de las clases static, otro problema está en el operador **this.** El problema es que al usar **this** dentro de una clase interna, this se refiere al objeto de la clase interna (es decir **this** dentro de *Motor* se refiere al objeto *Motor*). Para poder referirse al objeto contenedor (al coche) se usa *Clase*.**this** (*Coche*.**this**). Ejemplo:

```
public class Coche {
    public int velocidad;
    public int cilindrada;
    public Motor motor;

    public Coche(int cil) {
        motor=new Motor(cil);
        velocidad=0;
    }

    public class Motor{
        public int cilindrada;
        public Motor(int cil){
            Coche.this.cilindrada=cil;//Coche
             this.cilindrada=cil;//Motor
        }
    }
}
```

Por último las clases internas pueden ser anónimas (se verán más adelante al estar más relacionadas con interfaces y adaptadores).

interfaces

La limitación de que sólo se puede heredar de una clase, hace que haya problemas ya que muchas veces se deseará heredar de varias clases. Aunque ésta no es la finalidad directa de las interfaces, sí que tiene cierta relación

Mediante interfaces se definen una serie de comportamientos de objeto. Estos comportamientos puede ser "implementados" en una determinada clase. No definen el tipo de objeto que es, sino lo que puede hacer (sus capacidades). Por ello lo normal es que el nombre de las interfaces terminen con el texto "**able"** (*configurable*, *modificable, cargable*).

Por ejemplo en el caso de la clase Coche, esta deriva de la superclase Vehículo, pero además puesto que es un vehículo a motor, puede implementar métodos de una interfaz llamada por ejemplo **arrancable.** Se dirá entonces que la clase Coche es *arrancable*.

utilizar interfaces

Para hacer que una clase utilice una interfaz, se añade detrás del nombre de la clase la palabra **implements** seguida del nombre del interfaz. Se pueden poner varios nombres de interfaces separados por comas (solucionando, en cierto modo, el problema de la herencia múltiple).

```
class Coche extends vehiculo implements arrancable {
  public void arrancar (){
        ....
  }
  public void detenerMotor(){
        ....
  }
```

Hay que tener en cuenta que la interfaz *arrancable* no tiene porque tener ninguna relación de herencia con la clase vehículo, es más se podría implementar el interfaz *arrancable* a una bomba de agua.

creación de interfaces

Una interfaz en realidad es una serie de **constantes y métodos abstractos**. Cuando una clase implementa un determinado interfaz **debe** anular los métodos abstractos de éste, redefiniéndolos en la propia clase. Esta es la base de una interfaz, en realidad no hay una relación sino que hay una obligación por parte de la clase que implemente la interfaz de redefinir los métodos de ésta.

Una interfaz se crea exactamente igual que una clase (se crean en archivos propios también), la diferencia es que la palabra **interface** sustituye a la palabra **class** y que **sólo se pueden definir en un interfaz constantes y métodos abstractos**.

Todas las interfaces son abstractas y sus métodos también son todos abstractos y públicos (no hace falta poner el modificar **abstract** se toma de manera implícita). Las variables se tienen obligatoriamente que inicializar. Ejemplo:

```
interface arrancable(){
  boolean motorArrancado=false;
  void arrancar();
  void detenerMotor();
}
```

Los métodos son simples prototipos y toda variable se considera una constante (a no ser que se redefina en una clase que implemente esta interfaz, lo cual no tendría mucho sentido).

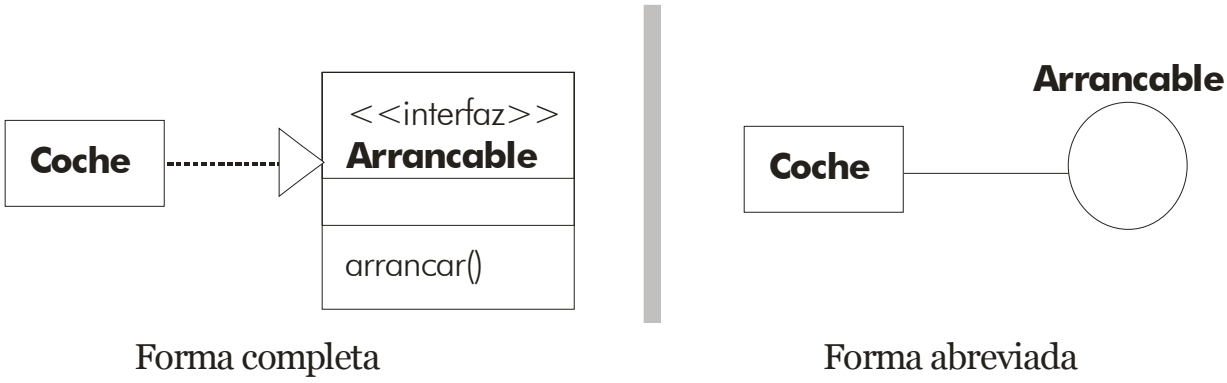


Ilustración 13, Diagramas de clases UML sobre la interfaz *Arrancable*

subinterfaces

Una interfaz puede heredarse de otra interfaz, como por ejemplo en:

```
interface dibujable extends escribible, pintable {
```

dibujable es subinterfaz de *escribible* y *pintable*. Es curioso, pero los interfaces sí admiten herencia múltiple. Esto significa que la clase que implemente el interfaz *dibujable* deberá incorporar los métodos definidos en *escribible* y *pintable*.

variables de interfaz

Al definir una interfaz, se pueden crear después variables de interfaz. Se puede interpretar esto como si el interfaz fuera un tipo especial de datos (que no de clase). La ventaja que proporciona es que pueden asignarse variables interfaz a cualquier objeto de una clase que implementa la interfaz. Esto permite cosas como:

```
Arrancable motorcito; //motorcito es una variable de tipo
                      // arrancable
Coche c=new Coche(); //Objeto de tipo coche
BombaAgua ba=new BombaAgua(); //Objeto de tipo BombaAgua
motorcito=c; //Motorcito apunta a c
motorcito.arrancar() //Se arrancará c
motorcito=ba; //Motorcito apunta a ba
motorcito=arrancar; //Se arranca la bomba de agua
```

El juego que dan estas variables es impresionante, debido a que fuerzan acciones sobre objetos de todo tipo, y sin importar este tipo; siempre y cuando estos objetos pertenezcan a clases que implementen el interfaz.

interfaces como funciones de retroinvocación

En C++ una función de retroinvocación es un puntero que señala a un método o a un objeto. Se usan para controlar eventos. En Java se usan interfaces para este fin.

Ejemplo:

```
interface Escribible {
   void escribe(String texto);
}

class Texto implements Escribible {
   ...
   public void escribe(texto){
         System.out.println(texto);
   }
}

class Prueba {
   Escribible escritor;
   public Prueba(Escribible e){
         escritor=e;
   }
   public void enviaTexto(String s){
         escritor.escribe(s);
   }
}
```

En el ejemplo *escritor* es una variable de la interfaz *Escribible*, cuando se llama a su método *escribe*, entonces se usa la implementación de la clase *texto*.

creación de paquetes

Un paquete es una colección de clases e interfaces relacionadas. El compilador de Java usa los paquetes para organizar la compilación y ejecución. Es decir, un paquete es una biblioteca. De hecho el nombre completo de una clase es el nombre del paquete en el que está la clase, punto y luego el nombre de la clase. Es decir si la clase *Coche* está dentro del paquete *locomoción*, el nombre completo de Coche es ***locomoción.Coche.***

A veces resulta que un paquete está dentro de otro paquete, entonces habrá que indicar la ruta completa a la clase. Por ejemplo ***locomoción.motor.Coche***

Mediante el comando **import** (visto anteriormente), se evita tener que colocar el nombre completo. El comando **import** se coloca antes de definir la clase. Ejemplo:

```
import locomoción.motor.Coche;
```

Gracias a esta instrucción para utilizar la clase Coche no hace falta indicar el paquete en el que se encuentra, basta indicar sólo *Coche*. Se puede utilizar el símbolo asterisco como comodín.

Ejemplo:

```
import locomoción.*;
//Importa todas las clase del paquete locomoción
```

Esta instrucción no importa el contenido de los paquetes interiores a *locomoción* (es decir que si la clase *Coche* está dentro del paquete *motor*, no sería importada con esa instrucción, ya que el paquete motor no ha sido importado, sí lo sería la clase *locomoción.BarcoDeVela*). Por ello en el ejemplo lo completo sería:

```
import locomoción.*;
import locomoción.motor.*;
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta en el paquete en el que está colocada la clase y, sino se encuentra, en los paquetes que se han importado al programa. Si ese nombre de clase se ha definido en un solo paquete, se usa. Si no es así podría haber **ambigüedad** por ello se debe usar un prefijo delante de la clase con el nombre del paquete.

Es decir:

```
paquete.clase
```

O incluso:

```
paquete1.paquete2......clase
```

En el caso de que el paquete sea subpaquete de otro más grande.

Las clases son visibles en el mismo paquete a no ser que se las haya declarado con el modificador **private.**

organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello es necesario usar la variable de entorno **CLASSPATH** de la línea de comandos. Esta variable se suele definir en el archivo **autoexec.bat** o en MI PC en el caso de las últimas versiones de Windows (Véase proceso de compilación, página 9). Hay que añadirla las rutas a las carpetas que contienen los paquetes (normalmente todos los paquetes se suelen crear en la misma carpeta), a estas carpetas se las llama **filesystems.**

Así para el paquete **prueba.reloj** tiene que haber una carpeta prueba, dentro de la cual habrá una carpeta reloj y esa carpeta prueba tiene que formar parte del **classpath.**

Una clase se declara perteneciente aun determinado paquete usando la instrucción **package** al principio del código (sin usar esta instrucción, la clase no se puede compilar). Si se usa package tiene que ser la primera instrucción del programa Java:

```
//Clase perteneciente al paquete tema5 que está en ejemplos
package ejemplos.tema5;
```

En los entornos de desarrollo o IDEs (NetBeans, JBuilder,...) se puede uno despreocupar de la variable classpath ya que poseen mecanismos de ayuda para gestionar los paquetes. Pero hay que tener en cuenta que si se compila a mano mediante el comando **java** (véase proceso de compilación, página i) se debe añadir el modificador **-cp** para que sean accesibles las clases contenidas en paquetes del classpath (por ejemplo **java -cp prueba.java).**

El uso de los paquetes permite que al compilar sólo se compile el código de la clase y de las clases importadas, en lugar de compilar todas las librerías. Sólo se compila lo que se utiliza.

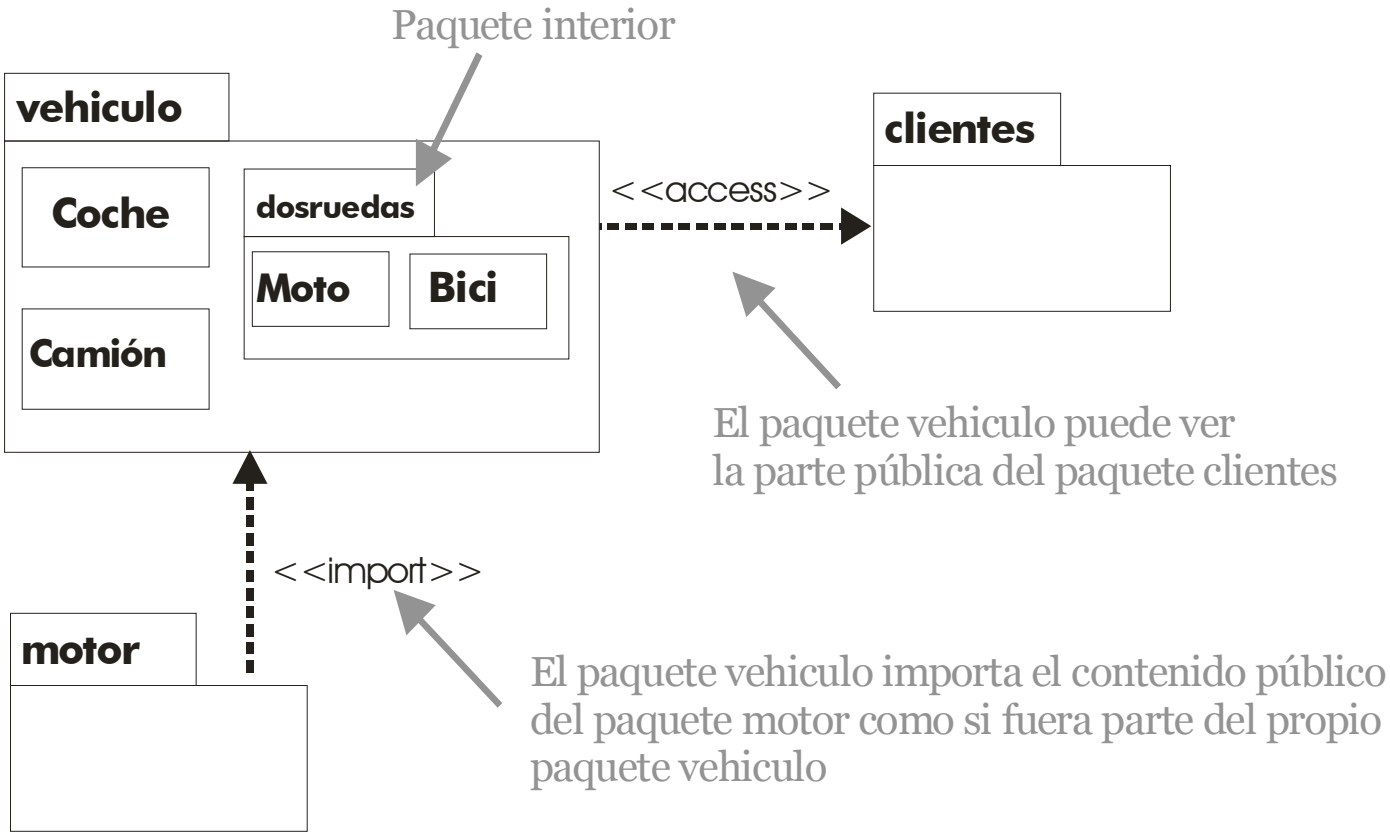


Ilustración 14, diagrama de paquetes UML

excepciones

introducción a las excepciones

Uno de los problemas más importantes al escribir aplicaciones es el tratamiento de los errores. Errores no previstos que distorsionan la ejecución del programa. Las **excepciones** de Java hacen referencia e este hecho. Se denomina excepción a una situación que no se puede resolver y que provoca la detención del programa; es decir una condición de error en tiempo de ejecución (es decir cuando el programa ya ha sido compilado y se está ejecutando). Ejemplos:

- El archivo que queremos abrir no existe
- Falla la conexión a una red
- La clase que se desea utilizar no se encuentra en ninguno de los paquetes reseñados con **import**

Los errores de sintaxis son detectados durante la compilación. Pero las excepciones pueden provocar situaciones irreversibles, su control debe hacerse en tiempo de ejecución y eso presenta un gran problema. En Java se puede preparar el código susceptible a provocar errores de ejecución de modo que si ocurre una excepción, el código es *lanzado (throw)* a una determinada rutina previamente preparada por el programador, que permite manipular esa excepción. Si la excepción no fuera capturada, la ejecución del programa se detendría irremediablemente.

En Java hay muchos tipos de excepciones (de operaciones de entrada y salida, de operaciones irreales. El paquete **java.lang.Exception** y sus subpaquetes contienen todos los tipos de excepciones.

Cuando se produce un error se genera un objeto asociado a esa excepción. Este objeto es de la clase **Exception** o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto Exception.

Hay una clase, la **java.lang.Error** y sus subclases que sirven para definir los errores irrecuperables más serios. Esos errores causan parada en el programa, por lo que el programador no hace falta que los manipule. Estos errores les produce el sistema y son incontrolables para el programador. Las excepciones son fallos más leves, y más manipulables.

try y catch

Las sentencias que tratan las excepciones son **try** y **catch.** La sintaxis es:

```
try {
        instrucciones que se ejecutan salvo que haya un error
}
catch (ClaseExcepción objetoQueCapturaLaExcepción) {
        instrucciones que se ejecutan si hay un error}
```

Puede haber más de una sentencia **catch** para un mismo bloque **try.** Ejemplo:

```
try {
        readFromFile("arch");
        ...
}
catch(FileNotFoundException e) {
        //archivo no encontrado
...
}
catch (IOException e) {
...
}
```

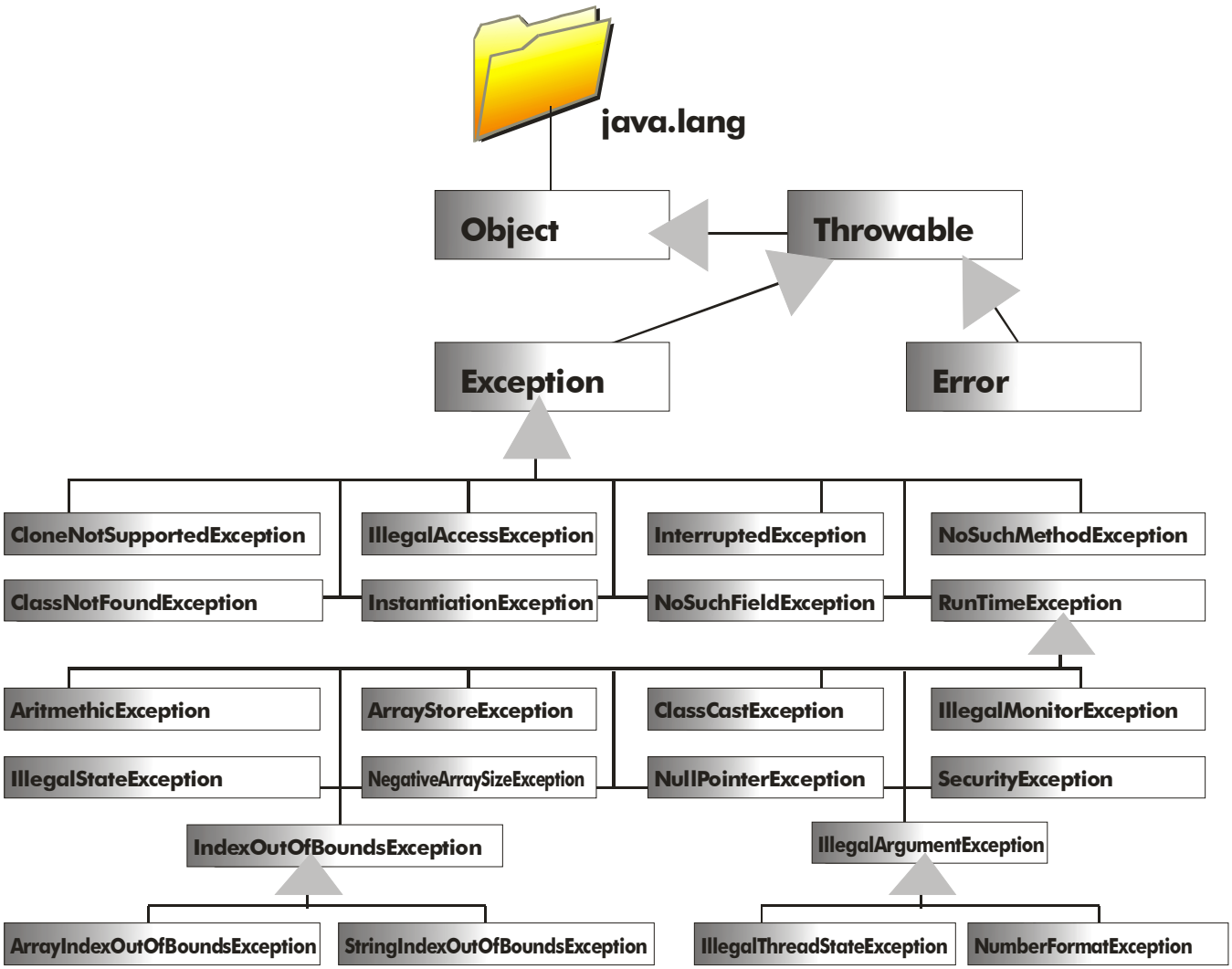


Ilustración 15, Jerarquía de las clases de manejo de excepciones

Dentro del bloque **try** se colocan las instrucciones susceptibles de provocar una excepción, el bloque **catch** sirve para capturar esa excepción y evitar el fin de la ejecución del programa. Desde el bloque catch se maneja, en definitiva, la excepción.

Cada catch maneja un tipo de excepción. Cuando se produce una excepción, se busca el catch que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase **Exception** es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo **AritmethicException** y el primer **catch** captura el tipo genérico **Exception**, será ese catch el que se ejecute y no los demás.

Por eso el último catch debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo **catch** que capture objetos Exception.

manejo de excepciones

Siempre se debe controlar una excepción, de otra forma nuestro software está a merced de los fallos. En la programación siempre ha habido dos formas de manejar la excepción:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que dio pies al error, se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

La filosofía de Java es del tipo **interrupción**, pero se puede intentar emular la reanudación encerrando el bloque **try** en un **while** que se repetirá hasta que el error deje de existir. Ejemplo:

```
boolean indiceNoValido=true;
int i; //Entero que tomará nos aleatorios de 0 a 9
String texto[]={"Uno","Dos","Tres","Cuatro","Cinco"};
while(indiceNoValido){
   try{
         i=Math.round(Math.random()*9);
         System.out.println(texto[i];
         indiceNoValido=false;
   }catch(ArrayIndexOutOfBoundsException exc){
         System.out.println("Fallo en el índice");
   }
}
```

En el código anterior, el índice *i* calcula un número del 0 al 9 y con ese número el código accede al array *texto* que sólo contiene 5 elementos. Esto producirá muy a menudo una excepción del tipo *ArrayIndexOutOfBoundsException* que es manejada por el *catch*

correspondiente. Normalmente no se continuaría intentando. Pero como tras el bloque *catch* está dentro del **while**, se hará otro intento y así hasta que no haya excepción, lo que provocará que *indiceNovalido* valga *true* y la salida, al fin, del while.

Como se observa en la **¡Error! No se encuentra el origen de la referencia.**, la clase **Exception** es la superclase de todos los tipos de excepciones. Esto permite utilizar una serie de métodos comunes a todas las clases de excepciones:

- **String getMessage().** Obtiene el mensaje descriptivo de la excepción o una indicación específica del error ocurrido:

```
try{
   ....
} catch (IOException ioe){
   System.out.println(ioe.getMessage());
}
```

- **String toString().** Escribe una cadena sobre la situación de la excepción. Suele indicar la clase de excepción y el texto de **getMessage().**

- **void printStackTrace()**. Escribe el método y mensaje de la excepción (la llamada información de pila). El resultado es el mismo mensaje que muestra el ejecutor (la máquina virtual de Java) cuando no se controla la excepción.

throws

Al llamar a métodos, ocurre un problema con las excepciones. El problema es, si el método da lugar a una excepción, ¿quién la maneja? ¿El propio método?¿O el código que hizo la llamada al método?

Con lo visto hasta ahora, sería el propio método quien se encargara de sus excepciones, pero esto complica el código. Por eso otra posibilidad es hacer que la excepción la maneje el código que hizo la llamada.

Esto se hace añadiendo la palabra **throws** tras la primera línea de un método. Tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el catch apropiado para esa excepción. Ejemplo:

```
void usarArchivo (String archivo) throws IOException,
InterruptedException {...
```

En este caso se está indicando que el método *usarArchivo* puede provocar excepciones del tipo *IOException* y *InterruptedException*. Esto significará, además, que el que utilice este método debe preparar el *catch* correspondiente para manejar los posibles errores.

Ejemplo:

```
try{
   ...
   objeto.usarArchivo("C:\texto.txt");//puede haber excepción
   ..
}
catch(IOException ioe){...
}
catch(InterruptedException ie){...
}
...//otros catch para otras posibles excepciones
```

throw

Esta instrucción nos permite lanzar a nosotros nuestras propias excepciones (o lo que es lo mismo, crear artificialmente nosotros las excepciones). Ante:

```
throw new Exception();
```

El flujo del programa se dirigirá a la instrucción *try/catch* más cercana. Se pueden utilizar constructores en esta llamada (el formato de los constructores depende de la clase que se utilice):

```
throw new Exception("Error grave, grave");
```

Eso construye una excepción con el mensaje indicado.

throw permite también *relanzar* excepciones. Esto significa que dentro de un catch podemos colocar una instrucción **throw** para lanzar la nueva excepción que será capturada por el catch correspondiente:

```
try{
   ...
} catch(ArrayIndexOutOfBoundsException exc){
   throw new IOException();
} catch(IOException){
   ...
}
```

El segundo catch capturará también las excepciones del primer tipo

finally

La cláusula finally está pensada para limpiar el código en caso de excepción. Su uso es:

```
try{
   ...
```

```
}catch (FileNotFoundException fnfe){
   ...
}catch(IOException ioe){
   ...
}catch(Exception e){
   ...
}finally{
   ...//Instrucciones de limpieza
}
```

Las sentencias finally se ejecutan tras haberse ejecutado el catch correspondiente. Si ningún catch capturó la excepción, entonces se ejecutarán esas sentencias antes de devolver el control al siguiente nivel o antes de romperse la ejecución.

Hay que tener muy en cuenta que **las sentencias finally se ejecutan independientemente de si hubo o no excepción**. Es decir esas sentencias se ejecutan siempre, haya o no excepción. Son sentencias a ejecutarse en todo momento. Por ello se coloca en el bloque finally código común para todas las excepciones (y también para cuando no hay excepciones.

clases fundamentales (I)

la clase *Object*

Todas las clases de Java poseen una superclase común, esa es la clase **Object**. Por eso los métodos de la clase Object son fundamentales ya que todas las clases los heredan. Esos métodos están pensados para todas las clases, pero hay que redefinirlos para que funcionen adecuadamente.

Es decir, Object proporciona métodos que son heredados por todas las clase. La idea es que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes como comprobar igualdad, clonar, ... y para ello habrá que redefinir esos métodos a fin de que se ajusten adecuadamente a cada clase.

comparar objetos

La clase Object proporciona un método para comprobar si dos objetos son iguales. Este método es **equals.** Este método recibe como parámetro un objeto con quien comparar y devuelve **true** si los dos objetos son iguales.

No es lo mismo equals que usar la comparación de igualdad. Ejemplos:

```
Coche uno=new Coche("Renault","Megane","P4324K");
Coche dos=uno;
boolean resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado también valdrá true
dos=new Coche("Renault","Megane","P4324K");
resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado ahora valdrá false
```

En el ejemplo anterior **equals** devuelve **true** si los dos coches tienen el mismo modelo, marca y matrícula . El operador "==" devuelve **true** si los dos objetos se refieren a la misma cosa (las dos referencias apuntan al mismo objeto).

Realmente en el ejemplo anterior la respuesta del método equals sólo será válida si en la clase que se está comparando (*Coche* en el ejemplo) se ha redefinido el método equals. Esto no es opcional sino **obligatorio si se quiere usar este método.** El resultado de equals depende de cuándo consideremos nosotros que devolver verdadero o falso. En el ejemplo anterior el método equals sería:

```
public class Coche extends Vehículo{
  public boolean equals (Object o){
        if ((o!=null) && (o instanceof Coche)){
              if (((Coche)o).matricula==matricula &&
                ((Coche)o).marca==marca
                    && ((Coche)o).modelo==modelo))
                return true
        }
        return false; //Si no se cumple todo lo anterior
```

Es necesario el uso de **instanceOf** ya que equals puede recoger cualquier objeto Object. Para que la comparación sea válida primero hay que verificar que el objeto es un coche. El argumento **o** siempre hay que convertirlo al tipo Coche para utilizar sus propiedades de Coche.

código hash

El método **hashCode()** permite obtener un número entero llamado **código hash.** Este código es un entero único para cada objeto que se genera aleatoriamente según su contenido. No se suele redefinir salvo que se quiera anularle para modificar su función y generar códigos hash según se desee.

clonar objetos

El método **clone** está pensado para conseguir una copia de un objeto. Es un método **protected** por lo que sólo podrá ser usado por la propia clase y sus descendientes, salvo que se le redefina con **public**.

Además si una determinada clase desea poder clonar sus objetos de esta forma, debe implementar la interfaz **Cloneable** (perteneciendo al paquete java.lang), que no contiene ningún método pero sin ser incluída al usar clone ocurriría una excepción del tipo **CloneNotSupportedException.** Esta interfaz es la que permite que el objeto sea clonable.

Ejemplo:

```
public class Coche extends Vehiculo implements arrancable,
Cloneable{
  public Object clone(){
        try{
              return (super.clone());
        }catch(CloneNotSupportedException cnse){
          System.out.println("Error inesperado en clone");
          return null;
        }
}
....
//Clonación
        Coche uno=new Coche();
        Coche dos=(Coche)uno.clone();
```

En la última línea del código anterior, el cast "(Coche)" es obligatorio ya que clone devuelve forzosamente un objeto tipo Object. Aunque este código generaría dos objetos distintos, el código hash sería el mismo.

método toString

Este es un método de la clase Object que da como resultado un texto que describe al objeto. la utiliza, por ejemplo el método **println** para poder escribir un método por pantalla. Normalmente en cualquier clase habría que definir el método **toString**. Sin redefinirlo el resultado podría ser:

```
Coche uno=new Coche();
System.out.println(uno);//Escribe: Coche@26e431
```

Si redefinimos este método en la clase Coche:

```
public String toString(){
   return("Velocidad :"+velocidad+"\nGasolina: "+gasolina);
}
```

Ahora en el primer ejemplo se escribiría la velocidad y la gasolina del coche.

lista completa de métodos de la clase Object

método	significado
protected Object **clone()**	Devuelve como resultado una copia del objeto.
boolean **equals(**Object obj**)**	Compara el objeto con un segundo objeto que es pasado como referencia (el objeto *obj*). Devuelve **true** si son iguales.
protected **void finalize()**	Destructor del objeto
Class **getClass()**	Proporciona la clase del objeto
int **hashCode()**	Devuelve un valor *hashCode* para el objeto
void **notify()**	Activa un hilo (thread) sencillo en espera.
void **notifyAll()**	Activa todos los hilos en espera.
String **toString()**	Devuelve una cadena de texto que representa al objeto
void **wait()**	Hace que el hilo actual espere hasta la siguiente notificación
void **wait(long** tiempo**)**	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo
void **wait(**long tiempo, int nanos**)**	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo o hasta que otro hilo interrumpa al actual

clase *Class*

La clase ***Object*** posee un método llamado **getClass()** que devuelve la clase a la que pertenece un determinado objeto. La clase **Class** es también una superclase común a todas las clase, pero a las clases que están en ejecución.

Class tiene una gran cantidad de métodos que permiten obtener diversa información sobre la clase de un objeto determinado en tiempo de ejecución. A eso se le llama **reflexión** (obtener información sobre el código de un objeto).

Ejemplo:

```
String prueba=”Hola, hola”;
Class clase=prueba.getClass();
System.out.println(clase.getName());//*java.lang.String*
System.out.println(clase.getPackage());//*package java.lang*
System.out.println(clase.getSuperclass());
                    //*class package java.lang.Object*
Class clase2= Class.forName(“java.lang.String”);
```

lista de métodos de *Class*

método	significado
String getName()	Devuelve el nombre completo de la clase.
String getPackage()	Devuelve el nombre del paquete en el que está la clase.
static Class forName(String s) throws ClassNotFoundException	Devuelve la clase a la que pertenece el objeto cuyo nombre se pasa como argumento. En caso de no encontrar el nombre lanza un evento del tipo **ClassNotFoundException.**
Class[] getClasses()	Obtiene un array con las clases e interfaces miembros de la clase en la que se utilizó este método.
ClassLoader getClassLoade()	Obtiene el **getClassLoader()** (el cargador de clase) de la clase
Class getComponentType()	Devuelve en forma de objeto **class,** el tipo de componente de un array (si la clase en la que se utilizó el método era un array). Por ejemplo devuelve **int** si la clase representa un array de tipo **int**. Si la clase no es un array, devuelve **null.**
Constructor getConstructor(Class[] parameterTypes) throws NoSuchMethodException, SecurityException	Devuelve el constructor de la clase que corresponde a lista de argumentos en forma de array **Class.**
Constructor[] getConstructors() throws SecurityException	Devuelve un array con todos los constructores de la clase.
Class[] getDeclaredClasses() throws SecurityException	Devuelve un array con todas las clases e interfaces que son miembros de la clase a la que se refiere este método. Puede provocar una excepción **SecurityException** si se nos deniega el acceso a una clase.
Constructor getDeclaredConstructor (Class[] parametros)	Obtiene el constructor que se corresponde a la lista de parámetros pasada en forma de array de objetos **Class.**
Constructor [] getDeclaredConstructors() throws NoSuchMethodException, SecurityException	Devuelve todos los constructores de la clase en forma de array de constructores.

método	significado
static Class forName(String nombre**)** throws ClassNotFoundException	
Field getDeclaredField(String nombre) **throws** NoSuchFieldException, SecurityException	Devuelve la propiedad declarada en la clase que tiene como nombre la cadena que se pasa como argumento.
Field [] getDeclaredFields() throws SecurityException	Devuelve todas las propiedades de la clase en forma de array de objetos **Field**.
Method getDeclaredMethod (String nombre, **Class[]** TipoDeParametros) **throws** NoSuchMethodException, SecurityException	Devuelve el método declarado en la clase que tiene como nombre la cadena que se pasa como argumento como tipo de los argumentos, el que indique el array **Class** especificado
Method [] getDeclaredMethods() throws `SecurityException`	Devuelve todos los métodos de la clase en forma de array de objetos **Method**.
Class getDeclaringClass() throws SecurityException	Devuelve la clase en la que se declaró la actual. Si no se declaró dentro de otra, devuelve **null**.
Field getField(String nombre**) throws** NoSuchFieldException, SecurityException	Devuelve (en forma de objeto **Field**) la propiedad pública cuyo nombre coincida con el que se pasa como argumento.
Field[] getFields() throws `SecurityException`	Devuelve un array que contiene una lista de todas las propiedades públicas de la clase.
Class[] getInterface()	Devuelve un array que representa a todos los interfaces que forman parte de la clase.
Method getMethod (String nombre, **Class[]** TipoDeParametros) **throws** NoSuchMethodException, SecurityException	Devuelve el método público de la clase que tiene como nombre la cadena que se pasa como argumento como tipo de los argumentos, el que indique el array **Class** especificado
Method [] getMethods() throws ***`SecurityException`***	Devuelve todos los métodos públicos de la clase en forma de array de objetos **Method**.
int getModifiers()	Devuelve, codificados, los modificadores de la clase (**protected, public,...**). Para decodificarlos hace falta usar la clase **Modifier.**
String getName()	Devuelve el nombre de la clase.
String getPackage()	Devuelve el paquete al que pertenece la clase.
ProtectionDomain getProtectionDomain()	Devuelve el dominio de protección de la clase. (JDK 1.2)
URL getResource(String nombre)	Devuelve, en forma de URL, el recurso cuyo nombre se indica
InputStream getResourceAsStream(String nombre)	Devuelve, en forma de InputStream, el recurso cuyo nombre se indica
class getSuperclass()	Devuelve la superclase a la que pertenece ésta. Si no hay superclase, devuelve **null**

método	significado
boolean isArray()	Devuelve **true** si la clase es un array
boolean isAssignableFrom(Class clas2**)**	Devuelve **true** si la clase a la que pertenece *clas2* es asignable a la clase actual.
boolean isInstance(Object *o***)**	Devuelve **true** si el objeto *o* es compatible con la clase. Es el equivalente dinámico al operador **instanceof.**
boolean isInterface()	Devuelve **true** si el objeto **class** representa a una interfaz.
boolean isPrimitive()	Devuelve **true** si la clase no tiene superclase.
Object newInstance() **throws** InstantiationException, IllegalAccessException	Crea un nuevo objeto a partir de la clase actual. El objeto se crea usando el constructor por defecto.
String toString()	Obtiene un texto descriptivo del objeto. Suele ser lo mismo que el resultado del método **getName().**

reflexión

En Java, por traducción del término **reflection,** se denomina reflexión a la capacidad de un objeto de examinarse a sí mismo. En el paquete **java.lang.reflect** hay diversas clases que tienen capacidad de realizar este examen. Casi todas estas clases han sido referenciadas al describir los métodos de la clase **Class.**

Class permite acceder a cada elemento de reflexión de una clase mediante dos pares de métodos. El primer par permite acceder a los métodos públicos (**getField** y **getFields** por ejemplo), el segundo par accede a cualquier elemento miembro (**getDeclaredField** y **getDeclaredFields**) por ejemplo.

clase Field

La clase java.lang.reflection.Field, permite acceder a las propiedades (campos) de una clase. Métodos interesantes:

método	significado
Object get ()	Devuelve el valor del objeto **Field**.
Class getDeclaringClass()	Devuelve la clase en la que se declaro la propiedad.
int getModifiers()	Devuelve, codificados, los modificadores de la clase (**protected, public,...**). Para decodificarlos hace falta usar la clase **Modifier.**
String getName()	Devuelve el nombre del campo.
Class getType()	Devuelve, en forma de objeto **Class**, el tipo de la propiedad.
void set(Object o, **Object** value**)**	Asigna al objeto un determinado valor.
String toString()	Cadena que describe al objeto.

clase Method

Representa métodos de una clase. Sus propios métodos son:

método	significado
Class getDeclaringClass()	Devuelve la clase en la que se declaro la propiedad.
Class[] getExceptionTypes()	Devuelve un array con todos los tipos de excepción que es capaz de lanzar el método.
int getModifiers()	Devuelve, codificados, los modificadores de la clase (**protected, public,...**). Para decodificarlos hace falta usar la clase **Modifier.**
String getName()	Devuelve el nombre del método.
Class getParameterTypes()	Devuelve, en forma de array **Class**, los tipos de datos de los argumentos del método.
Class getReturnType()	Devuelve, en forma de objeto **Class**, el tipo de datos que devuelve el método.
void invoke(Object o, **Object[]** argumentos**)**	Invoca al método *o* usando la lista de parámetros indicada.
String toString()	Cadena que describe al objeto.

clase Constructor

Representa constructores. Tiene casi los mismos métodos de la clase anterior.

método	significado
Class getDeclaringClass()	Devuelve la clase en la que se declaro la propiedad.
Class[] getExceptionTypes()	Devuelve un array con todos los tipos de excepción que es capaz de lanzar el método.
int getModifiers()	Devuelve, codificados, los modificadores de la clase (**protected, public,...**). Para decodificarlos hace falta usar la clase **Modifier.**
String getName()	Devuelve el nombre del método.
Class getParameterTypes()	Devuelve, en forma de array **Class**, los tipos de datos de los argumentos del método.
Object newInstance(Object[] argumentos**)** **throws** InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException	Crea un nuevo objeto usando el constructor de clase que se corresponda con la lista de argumentos pasada.
String toString()	Cadena que describe al objeto.

clases para tipos básicos

En Java se dice que todo es considerado un objeto. Para hacer que esta filosofía sea más real se han diseñado una serie de clases relacionadas con los tipos básicos. El nombre de estas clases es:

clase	representa al tipo básico..
java.lang.**Void**	**void**
java.lang.**Boolean**	**boolean**
java.lang.**Character**	**char**
java.lang.**Byte**	**byte**
java.lang.**Short**	**short**
java.lang.**Integer**	**int**
java.lang.**Long**	**long**
java.lang.**Float**	**float**
java.lang.**Double**	**double**

Hay que tener en cuenta que no son equivalentes a los tipos básicos. La creación de estos tipos lógicamente requiere usar constructores, ya que son objetos y no tipos básicos.

```
Double n=new Double(18.3);
Double o=new Double("18.5");
```

El constructor admite valores del tipo básico relacionado e incluso valores String que contengan texto convertible a ese tipo básico. Si ese texto no es convertible, ocurre una excepción del tipo **NumberFormatException**.

La conversión de un String a un tipo básico es una de las utilidades básicas de estas clases, por ello estas clases poseen el método estático **valueOf** entre otros para convertir un String en uno de esos tipos. Ejemplos:

```
String s="2500";
Integer a=Integer.valueOf(s);
Short b=Short.valueOf(s);
Double c=Short.valueOf(s);
Byte d=Byte.valueOf(s);//Excepción!!!
```

Hay otro método en cada una de esas clases que se llama **parse**. La diferencia estriba en que en los métodos **parse** la conversión se realiza hacia tipos básicos (int, double, float, boolean,...) y no hacia las clase anteriores. Ejemplo:

```
String s="2500";
int y=Integer.parseInt(s);
short z=Short.parseShort(s);
double c=Short.parseDouble(s);
byte x=Byte.parseByte(s);
```

Estos métodos son todos estáticos. Todas las clases además poseen métodos dinámicos para convertir a otros tipos (intValue, longValue,... o el conocido toString).

Todos estos métodos lanzan excepciones del tipo **NumberFormatException,** que habrá que capturar con el try y el catch pertinentes.

Además han redefinido el método **equals** para comparar objetos de este tipo. Además poseen el método **compareTo** que permite comparar dos elementos de este tipo (este método se maneja igual que el compareTo de la clase String, ver comparación entre objetos String, página 35)

clase StringBuffer

La clase String tiene una característica que puede causar problemas, y es que los objetos String se crean cada vez que se les asigna o amplia el texto. Esto hace que la ejecución sea más lenta. Este código:

```
String frase="Esta ";
frase += "es ";
frase += "la ";
frase += "frase";
```

En este código se crean cuatro objetos String y los valores de cada uno son copiados al siguiente. Por ello se ha añadido la clase **StringBuffer** que mejora el rendimiento. La concatenación de textos se hace con el método **append:**

```
StringBuffer frase = new StringBuffer("Esta ");
frase.append("es ");
frase.append("la ");
frase.append("frase.");
```

Por otro lado el método **toString** permite pasar un **StringBuffer** a forma de cadena String.

```
StringBuffer frase1 = new StringBuffer("Valor inicial");
...
String frase2 = frase1.toString();
```

Se recomienda usar StringBuffer cuando se requieren cadenas a las que se las cambia el texto a menudo. Posee métodos propios que son muy interesantes para realizar estas modificaciones (**insert, delete, replace,...**).

métodos de StringBuffer

método	descripción
StringBuffer **append(***tipo* variable**)**	Añade al *StringBuffer* el valor en forma de cadena de la variable
char **charAt(**int pos**)**	Devuelve el carácter que se encuentra en la posición *pos*

método	descripción
int capacity()	Da como resultado la capacidad actual del *StringBuffer*
StringBuffer delete(int inicio, **int** fin)	Borra del *StringBuffer* los caracteres que van desde la posición *inicio* a la posición *fin*
StringBuffer deleteCharAt(int pos**)**	Borra del *StringBuffer* el carácter situado en la posición *pos*
void ensureCapacity(int capadMinima**)**	Asegura que la capacidad del *StringBuffer* sea al menos la dada en la función
void getChars(int srcInicio, **int** srcFin, **char[]** dst, **int** dstInicio**)**	Copia a un array de caracteres cuyo nombre es dado por el tercer parámetro, los caracteres del *StringBuffer* que van desde *srcInicio* a *srcFin*. Dichos caracteres se copiarán en el array desde la posición *dstInicio*
StringBuffer insert(int pos, ***tipo*** valor**)**	Inserta el valor en forma de cadena a partir de la posición *pos* del *StringBuffer*
int length()	Devuelve el tamaño del *StringBuffer*
StringBuffer replace(int inicio**, int** fin**, String** texto**)**	Reemplaza la subcadena del *StringBuffer* que va desde *inicio* a *fin* por el *texto* indicado
StringBuffer reverse()	Se cambia el *StringBuffer* por su inverso
void setLength(int tamaño**)**	Cambia el tamaño del *StringBuffer* al tamaño indicado.
String substring(int inicio**)**	Devuelve una cadena desde la posición inicio
String substring(int inicio, **int** fin**)**	Devuelve una cadena desde la posición *inicio* hasta la posición *fin*
String toString()	Devuelve el *StringBuffer* en forma de cadena **String**

números aleatorios

La clase **java.util.Random** está pensada para la producción de elementos aleatorios. Los números aleatorios producen dicha aleatoriedad usando una fórmula matemática muy compleja que se basa en, a partir de un determinado número obtener aleatoriamente el siguiente. Ese primer número es la semilla.

El constructor por defecto de esta clase crea un número aleatorio utilizando una semilla obtenida a partir de la fecha y la hora. Pero si se desea repetir continuamente la misma semilla, se puede iniciar usando un determinado número **long**:

```
Random r1=Random();//Semilla obtenida de la fecha y hora
Random r2=Random(182728L);//Semilla obtenida de un long
```

métodos de Random

método	devuelve
boolean nextBoolean()	**true** o **false** aleatoriamente
int nextInt()	un **int**

método	devuelve
int nextInt(int n**)**	Un número entero de 0 a *n-1*
long nextLong()	Un **long**
float nextFloat()	Número decimal de -1,0 a 1.0
double nextDouble()	Número doble de -1,0 a 1.0
void setSeed(long semilla**)**	Permite cambiar la semilla.

fechas

Sin duda alguna el control de fechas y horas es uno de los temas más pesados de la programación. Por ello desde Java hay varias clase dedicadas a su control.

La clase **java.util.Calendar** permite usar datos en forma de día mes y año, su descendiente **java.util.GregorianCalendar** añade compatibilidad con el calendario Gregoriano, la clase **java.util.Date** permite trabajar con datos que representan un determinado instante en el tiempo y la clase **java.text.DateFormat** está encargada de generar distintas representaciones de datos de fecha y hora.

clase Calendar

Se trata de una clase abstracta (no se pueden por tanto crear objetos Calendar) que define la funcionalidad de las fechas de calendario y define una serie de atributos estáticos muy importante para trabajar con las fechas. Entre ellos (se usan siempre con el nombre Calendar, por ejemplo Calendar.DAY_OF_WEEK):

- **Día de la semana: DAY_OF_WEEK** número del día de la semana (del 1 al 7). Se pueden usar las constantes MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY. Hay que tener en cuenta que usa el calendario inglés, con lo que el día 1 de la semana es el domingo (SUNDAY).
- **Mes: MONTH** es el mes del año (del 0, enero, al 11, diciembre). Se pueden usar las constantes: JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER.
- **Día del mes: DAY_OF_MONTH** número del día del mes (empezando por 1).
- **Semana del año: WEEK_OF_YEAR** indica o ajusta el número de semana del año.
- **Semana del mes: WEEK_OF_MONTH** indica o ajusta el número de semana del mes.
- **Día del año: DAY_OF_YEAR** número del día del año (de 1 a 365).
- **Hora: HOUR**, hora en formato de 12 horas. **HOUR_OF_DAY** hora en formato de 24 horas.
- **AM_PM.** Propiedad que sirve para indicar en qué parte del día estamos, AM o PM**.**
- **Minutos.** MINUTE

- **Segundos.** SECOND también se puede usar MILLISECOND para los milisegundos.

Esta clase también define una serie de métodos abstractos y estáticos.

clase GregorianCalendar

Es subclase de la anterior (por lo que hereda todos sus atributos). Permite crear datos de calendario gregoriano. Tiene numerosos constructores, algunos de ellos son:

```
GregorianCalendar fecha1=new GregorianCalendar();
  //Crea fecha1 con la fecha actual
GregorianCalendar fecha2=new GregorianCalendar(2003,7,2);
  //Crea fecha2 con fecha 2 de agosto de 2003
GregorianCalendar fecha3=new
GregorianCalendar(2003,Calendar.AUGUST,2);
  //Igual que la anterior
GregorianCalendar fecha4=new
GregorianCalendar(2003,7,2,12,30);
  //2 de Agosto de 2003 a las 12:30
GregorianCalendar fecha5=new
GregorianCalendar(2003,7,2,12,30,15);
  //2 de Agosto de 2003 a las 12:30:15
```

método get

El método **get** heredado de la clase **Calendar** sirve para poder obtener un detalle de una fecha. A este método se le pasa el atributo a obtener (véase lista de campos en la clase **Calendar**). Ejemplos:

```
GregorianCalendar fecha=new
  GregorianCalendar(2003,7,2,12,30,23);
System.out.println(fecha.get(Calendar.MONTH));
System.out.println(fecha.get(Calendar.DAY_OF_YEAR));
System.out.println(fecha.get(Calendar.SECOND));
System.out.println(fecha.get(Calendar.MILLISECOND));
/* La salida es
  7
  214
  23
  0
*/
```

método set

Es el contrario del anterior, sirve para modificar un campo del objeto de calendario. Tiene dos parámetros: el campo a cambiar (MONTH, YEAR,...) y el valor que valdrá ese campo:

```
fecha.set(Calendar.MONTH, Calendar.MAY);
fecha.set(Calendar.DAY_OF_MONTH, 12)
```

Otro uso de set consiste en cambiar la fecha indicando, año, mes y día y, opcionalmente, hora y minutos.

```
fecha.set(2003,17,9);
```

método getTime

Obtiene el objeto **Date** equivalente a al representado por el GregorianCalendar. En Java los objetos Date son fundamentales para poder dar formato a las fechas.

método setTime

Hace que el objeto de calendario tome como fecha la representada por un objeto date. Es el inverso del anterior

```
Date d=new Date()
GregorianCalendar gc=new GregorianCalendar()
g.setTime(d);
```

método getTimeInMillis

Devuelve el número de milisegundos que representa esa fecha.

clase Date

Representa una fecha en forma de milisegundos transcurridos, su idea es representar un instante. Cuenta fechas desde el 1900. Normalmente se utiliza conjuntamente con la clase **GregorianCalendar.** Pero tiene algunos métodos interesantes.

construcción

Hay varias formas de crear objetos Date:

```
Date fecha1=new Date();//Creado con la fecha actual
Date fecha2=(new GregorianCalendar(2004,7,6)).getTime();
```

método after

Se le pasa como parámetro otro objeto Date. Devuelve **true** en el caso de que la segunda fecha no sea más moderna. Ejemplo:

```
GregorianCalendar gc1=new GregorianCalendar(2004,3,1);
GregorianCalendar gc2=new GregorianCalendar(2004,3,10);
Date fecha1=gc1.getTime();
Date fecha2=gc2.getTime();
System.out.println(fecha1.after(fecha2));
//Escribe false porque la segunda fecha es más reciente
```

método before

Inverso al anterior. Devuelve **true** si la fecha que recibe como parámetro no es más reciente.

métodos equals y compareTo

Funcionan igual que en otros muchos objetos. **equals** devuelve **true** si la fecha con la que se compara es igual que la primera (incluidos los milisegundos). **compareTo** devuelve -1 si la segunda fecha es más reciente, 0 si son iguales y 1 si es más antigua.

clase DateFormat

A pesar de la potencia de las clases relacionadas con las fechas vistas anteriormente, sigue siendo complicado y pesado el hecho de hacer que esas fechas aparezcan con un formato más legible por un usuario normal.

La clase DateFormat nos da la posibilidad de formatear las fechas. Se encuentra en el paquete **java.text**. Hay que tener en cuenta que no representa fechas, sino maneras de dar formato a las fechas. Es decir un objeto DateFormat representa un formato de fecha (formato de fecha larga, formato de fecha corta,...).

creación básica

Por defecto un objeto DateFormat con opciones básicas se crea con:

```
DateFormat sencillo=DateFormat.getInstance();
```

Eso crea un objeto DateFormat con formato básico. **getInstance()** es un método estático de la clase **DateFormat** que devuelve un objeto DateFormat con formato sencillo.

el método format

Todos los objetos DateFormat poseen un método llamado **format** que da como resultado una cadena String y que posee como parámetro un objeto de tipo Date. El texto devuelto representa la fecha de una determinada forma. El formato es el indicado durante la creación del objeto DateFormat. Ejemplo:

```
Date fecha=new Date();//fecha actual
DateFormat df=DateFormat.getInstance();//Formato básico
System.out.println(df.format(fecha);
//Ejemplo de resultado: 14/04/04 10:37
```

creaciones de formato sofisticadas

El formato de fecha se puede configurar al gusto del programador. Esto es posible ya que hay otras formas de crear formatos de fecha. Todas las opciones consisten en utilizar los siguientes métodos estáticos (todos ellos devuelven un objeto DateFormat):

- **DateFormat.getDateInstance.** Crea un formato de fecha válido para escribir sólo la fecha; sin la hora.
- **DateFormat.getTimeInstance.** Crea un formato de fecha válido para escribir sólo la hora; sin la fecha.

- **DateFormat.getDateTimeInstance**. Crea un formato de fecha en el que aparecerán la fecha y la hora.

Todos los métodos anteriores reciben un parámetro para indicar el formato de fecha y de hora (el último método recibe dos: el primer parámetro se refiere a la fecha y el segundo a la hora). Ese parámetro es un número, pero es mejor utilizar las siguientes constantes estáticas:

- **DateFormat.SHORT.** Formato corto.
- **DateFormat.MEDIUM**. Formato medio
- **DateFormat.LONG .** Formato largo.
- **DateFormat.FULL**. Formato completo

Ejemplo:

```
DateFormat df=DateFormat.getDateInstance(DateFormat.LONG);
System.out.println(df.format(new Date()));
//14 de abril de 2004
DateFormat
  df2=DateFormat.getDateTimeInstance(DateFormat.LONG);
System.out.println(df2.format(new Date()));
// 14/04/04 00H52' CEST
```

La fecha sale con el formato por defecto del sistema (por eso sale en español si el sistema Windows está en español).

método parse

Inverso al método format. Devuelve un objeto Date a partir de un String que es pasado como parámetro. Este método lanza excepciones del tipo **ParseException** (clase que se encuentra en el paquete **java.text**), que estamos obligados a capturar. Ejemplo:

```
DateFormat df=DateFormat.getDateTimeInstance(
  DateFormat.SHORT,DateFormat.FULL);
try{
  fecha=df2.parse("14/3/2004 00H23' CEST");
}
catch(ParseException pe){
  System.out.println("cadena no válida");
}
```

Obsérvese que el contenido de la cadena debe ser idéntica al formato de salida del objeto DateFormat de otra forma se generaría la excepción.

Es un método muy poco usado.

cadenas delimitadas. *StringTokenizer*

introducción

Se denomina cadena delimitada a aquellas que contienen texto que está dividido en partes (***tokens***) y esas partes se dividen mediante un carácter (o una cadena) especial. Por ejemplo la cadena 7647-34-123223-1-234 está delimitada por el guión y forma 5 tokens.

Es muy común querer obtener cada zona delimitada, cada **token**, de la cadena. Se puede hacer con las clases que ya hemos visto, pero en el paquete **java.util** disponemos de la clase más apropiada para hacerlo, **StringTokenizer.**

Esa clase representa a una cadena delimitada de modo además que en cada momento hay un puntero interno que señala al siguiente *token* de la cadena. Con los métodos apropiados podremos avanzar por la cadena.

construcción

La forma común de construcción es usar dos parámetros: el texto delimitado y la cadena delimitadora. Ejemplo:

```
StringTokenizer st=new StringTokenizer("1234-5-678-9-00","-");
```

Se puede construir también el *tokenizer* sólo con la cadena, sin el delimitador. En ese caso se toma como delimitador el carácter de nueva línea (\n), el retorno de carro (\r), el tabulador (\t) o el espacio. Los *tokens* son considerados sin el delimitador (en el ejemplo sería 1234, 5, 678, 9 y 00, el guión no cuenta).

USO

Para obtener las distintas partes de la cadena se usan estos métodos:

- String **nextToken().** Devuelve el siguiente token. La primera vez devuelve el primer texto de la cadena hasta la llegada del delimitador. Luego devuelve el siguiente texto delimitado y así sucesivamente. Si no hubiera más tokens devuelve la excepción **NoSuchElementException**. Por lo que conviene comprobar si hay más tokens.
- boolean **hasMoreTokens().** Devuelve **true** si hay más tokens en el objeto **StringTokenizer.**
- int **countTokens().** Indica el número de tokens que quedan por obtener. El puntero de tokens no se mueve.

Ejemplo:

```
String tokenizada="10034-23-43423-1-3445";
StringTokenizer st=new StringTokenizer(tokenizada,"-");
while (st.hasMoreTokens()){
  System.out.println(st.nextToken());
}// Obtiene:10034 23 43423 1 y 3445
```

entrada y salida en Java

El paquete **java.io** contiene todas las clases relacionadas con las funciones de entrada (**input**) y salida (**output**). Se habla de E/S (o de I/O) refiriéndose a la entrada y salida. En términos de programación se denomina **entrada** a la posibilidad de introducir datos hacia un programa; **salida** sería la capacidad de un programa de mostrar información al usuario.

clases para la entrada y la salida

Java se basa en las secuencias para dar facilidades de entrada y salida. Cada secuencia es una corriente de datos con un emisor y un receptor de datos en cada extremo. Todas las clases relacionadas con la entrada y salida de datos están en el paquete **java.io.**

Los datos fluyen en serie, byte a byte. Se habla entonces de un **stream** (corriente de datos, o mejor dicho, corriente de bytes). Hay otro stream que lanza caracteres (tipo char Unicode, de dos bytes), se habla entonces de un reader (si es de lectura) o un writer (escritura).

Los problemas de entrada / salida suelen causar excepciones de tipo **IOException** o de sus derivadas. Con lo que la mayoría de operaciones deben ir inmersas en un **try.**

InputStream/ OutputStream

Clases **abstractas** que definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). Esas son corrientes de bits, no representan ni textos ni objetos. Poseen numerosas subclases, de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Aquí se definen los métodos **read()** (Leer) y **write()** (escribir). Ambos son métodos que trabajan con los datos, byte a byte.

Reader/Writer

Clases **abstractas** que definen las funciones básicas de escritura y lectura basada en caracteres Unicode. Se dice que estas clases pertenecen a la jerarquía de lectura/escritura orientada a caracteres, mientras que las anteriores pertenecen a la jerarquía orientada a bytes.

Aparecieron en la versión 1.1 y no substituyen a las anteriores. Siempre que se pueda es más recomendable usar clases que deriven de estas.

Posee métodos **read** y **write** adaptados para leer arrays de caracteres.

InputStreamReader/ OutputStreamWriter

Son clases que sirven para adaptar la entrada y la salida. El problema está en que las clases anteriores trabajan de forma muy distinta y ambas son necesarias. Por ello InputStreamReader convierte una corriente de datos de tipo InputStream a forma de Reader.

DataInputStream/DataOutputStream

Leen corrientes de datos de entrada en forma de byte, pero adaptándola a los tipos simples de datos (**int, short, byte,..., String)**. Tienen varios métodos read y write para leer y escribir datos de todo tipo. En el caso de DataInputStream son:

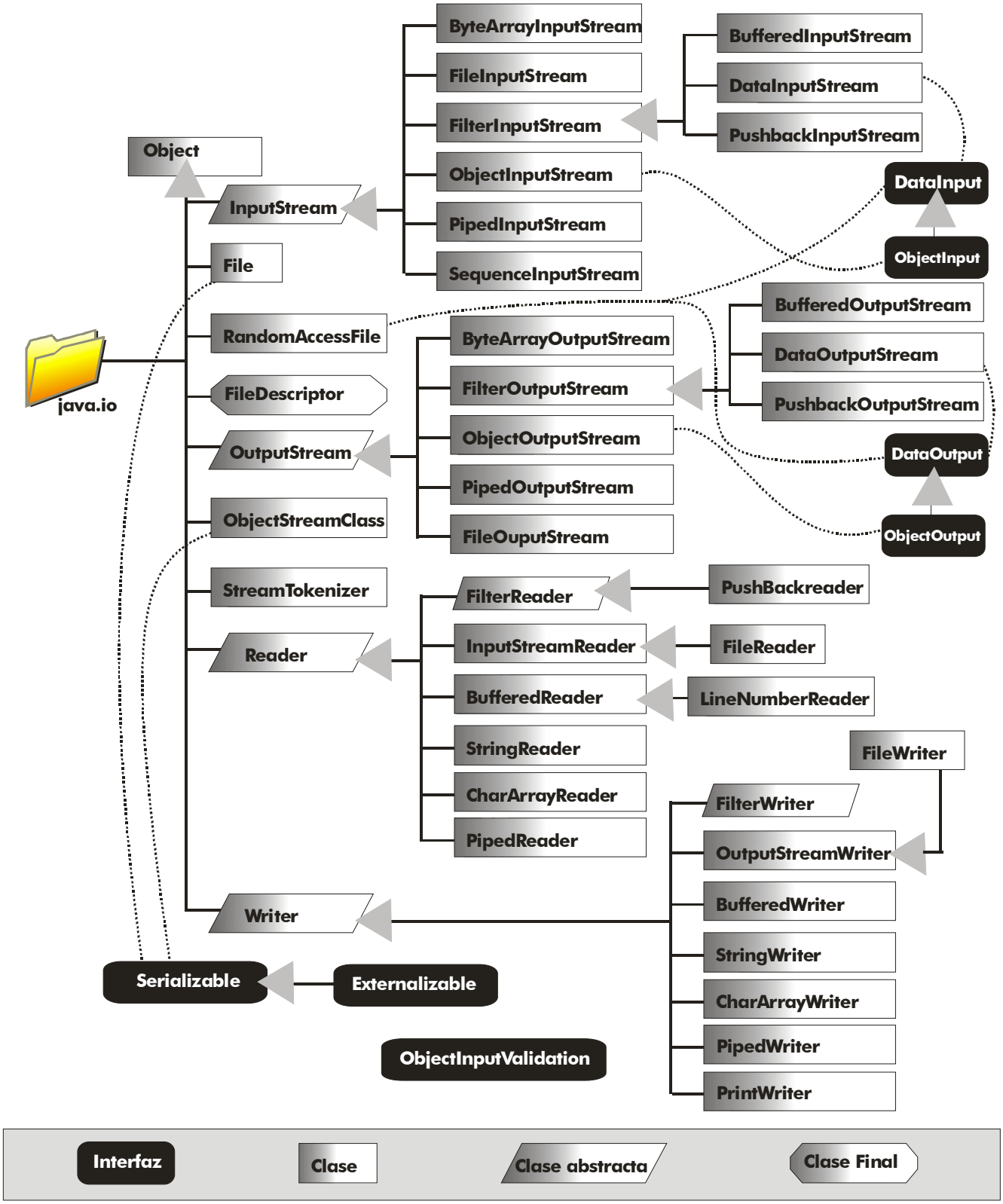


Ilustración 16, Clases e interfaces del paquete java.io

- **readBoolean().** Lee un valor booleano de la corriente de entrada. Puede provocar excepciones de tipo **IOException** o excepciones de tipo **EOFException**, esta última se produce cuando se ha alcanzado el final del archivo y es una excepción derivada de la anterior, por lo que si se capturan ambas, ésta debe ir en un **catch** anterior (de otro modo, el flujo del programa entraría siempre en la **IOException**).

- **readByte().** Idéntica a la anterior, pero obtiene un byte. Las excepciones que produce son las mismas

- **readChar, readShort, readInt, readLong, readFloat, readDouble**. Como las anteriores, pero leen los datos indicados.

- **readUTF()**. Lee un String en formato UTF (codificación norteamericana). Además de las excepciones comentadas antes, puede ocurrir una excepción del tipo **UTFDataFormatException (**derivada de **IOException**) si el formato del texto no está en UTF.

Por su parte, los métodos de DataOutputStream son:

- **writeBoolean, writeByte, writeDouble, writeFloat, writeShort, writeUTF, writeInt, writeLong.** Todos poseen un argumento que son los datos a escribir (cuyo tipo debe coincidir con la función).

ObjectInputStream/ObjectOutputStream

Filtros de secuencia que permiten leer y escribir objetos de una corriente de datos orientada a bytes. Sólo tiene sentido si los datos almacenados son objetos. Aporta un nuevo método de lectura:

- **readObject.** Devuelve un objeto Object de los datos de la entrada. En caso de que no haya un objeto o no sea serializable, da lugar a excepciones. Las excepciones pueden ser: **ClassNotFoundExcepcion, InvalidClassExcepcion, StreamCorruptedException, OptionalDataException** o **IOException** a secas.

La clase ObjectOutputStream posee el método de escritura de objetos **writeObject** al que se le pasa el objeto a escribir. Este método podría dar lugar en caso de fallo a excepciones **IOException, NotSerializableException** o **InvalidClassException.**.

BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter

La palabra **buffered** hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura. Los datos se almacenan en una memoria temporal antes de ser realmente leídos o escritos. Se trata de cuatro clase que trabajan con métodos distintos pero que suelen trabajar con las mismas corrientes de entrada que podrán ser de bytes (Input/OutputStream) o de caracteres (Reader/Writer).

La clase **BufferedReader** aporta el método **readLine** que permite leer caracteres hasta la presencia de **null** o del salto de línea.

PrintWriter

Secuencia pensada para impresión de texto. Es una clase escritora de caracteres en flujos de salida, que posee los métodos **print y println** ya comentados anteriormente, que otorgan gran potencia a la escritura.

FileInputStream/FileOutputStream/FileReader/FileWriter

Leen y escriben en archivos (File=Archivo).

PipedInputStream/PipedOutputStream

Permiten realizar canalizaciones entre la entrada y la salida; es decir lo que se lee se utiliza para una secuencia de escritura o al revés.

entrada y salida estándar

las clases in y out

java.lang.System es una clase que poseen multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están la clase **in** que es un **InputStream** que representa la entrada estándar (normalmente el teclado) y **out** que es un **OutputStream** que representa a la salida estándar (normalmente la pantalla). Hay también una clase **err** que representa a la salida estándar para errores. El uso podría ser:

```
InputStream stdin =System.in;
OutputStream stdout=System.out;
```

El método **read()** permite leer un byte. Este método puede lanzar excepciones del tipo **IOException** por lo que debe ser capturada dicha excepción.

```
int valor=0;
try{
  valor=System.in.read();
}
catch(IOException e){
  ...
}
System.out.println(valor);
```

No tiene sentido el listado anterior, ya que **read()** lee un byte de la entrada estándar, y en esta entrada se suelen enviar caracteres, por lo que el método read no es el apropiado. El método **read** puede poseer un argumento que es un array de bytes que almacenará cada carácter leído y devolverá el número de caracteres leído

```
InputStream stdin=System.in;
int n=0;
byte[] caracter=new byte[1024];
```

```
try{
   n=System.in.read(caracter);
}
catch(IOException e){
   System.out.println("Error en la lectura");
}
for (int i=0;i<=n;i++)
   System.out.print((char)caracter[i]);
```

El lista anterior lee una serie de bytes y luego los escribe. La lectura almacena el código del carácter leído, por eso hay que hacer una conversión a **char.**

Para saber que tamaño dar al array de bytes, se puede usar el método **available()** de la clase **InputStream** la tercera línea del código anterior sería:

```
byte[] carácter=new byte[System.in.available];
```

Conversión a forma de Reader

El hecho de que las clases **InputStream** y **OutputStream** usen el tipo byte para la lectura, complica mucho su uso. Desde que se impuso Unicode y con él las clases **Reader** y **Writer**, hubo que resolver el problema de tener que usar las dos anteriores.

La solución fueron dos clases: **InputStreamReader** y **OutputStreamWriter.** Se utilizan para convertir secuencias de byte en secuencias de caracteres según una determinada configuración regional. Permiten construir objetos de este tipo a partir de objetos **InputStream** u **OutputStream.** Puesto que son clases derivadas de **Reader** y **Writer** el problema está solucionado.

El constructor de la clase **InputStreamReader** requiere un objeto **InputStream** y, opcionalmente, una cadena que indique el código que se utilizará para mostrar caracteres (por ejemplo "ISO-8914-1" es el código Latín 1, el utilizado en la configuración regional). Sin usar este segundo parámetro se construye según la codificación actual (es lo normal).

Lo que hemos creado de esa forma es un objeto *convertidor*. De esa forma podemos utilizar la función read orientada a caracteres Unicode que permite leer caracteres extendidos. Está función posee una versión que acepta arrays de caracteres, con lo que la versión *writer* del código anterior sería:

```
InputStreamReader stdin=new InputStreamReader(System.in);
char caracter[]=new char[1024];
int numero=-1;
try{
     numero=stdin.read(caracter);
}
catch(IOException e){
     System.out.println("Error en la lectura");
}
for(int i=0;i<numero;i++)
     System.out.print(caracter[i]);
```

Lectura con readLine

El uso del método read con un array de caracteres sigue siendo un poco enrevesado. Por ello para leer cadenas de caracteres se suele utilizar la clase **BufferedReader**. La razón es que esta clase posee el método **ReadLine()** que permite leer una línea de texto en forma de String, que es más fácil de manipular. Esta clase usa un constructor que acepta objetos **Reader** (y por lo tanto **InputStreamReader**, ya que desciende de ésta) y, opcionalmente, el número de caracteres a leer.

Hay que tener en cuenta que el método *ReadLine* (como todos los métodos de lectura) puede provocar excepciones de tipo *IOException* por lo que, como ocurría con las otras lecturas, habrá que capturar dicha lectura.

```
String texto="";
try{
   //Obtención del objeto Reader
   InputStreamReader conv=new InputStreamReader(System.in);
   //Obtención del BufferedReader
   BufferedReader entrada=new BufferedReader(conv);
   texto=entrada.readLine();
}
catch(IOException e){
   System.out.println("Error");
}
System.out.println(texto);
```

Ficheros

Una aplicación Java puede escribir en un archivo, salvo que se haya restringido su acceso al disco mediante políticas de seguridad. La dificultad de este tipo de operaciones está en que los sistemas de ficheros son distintos en cada sistema y aunque Java intentar aislar la configuración específica de un sistema, no consigue evitarlo del todo.

clase File

En el paquete **java.io** se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

construcción de objetos de archivo

Utiliza como único argumento una cadena que representa una ruta en el sistema de archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

```
File archivo1=new File("/datos/bd.txt");
File carpeta=new File("datos");
```

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. La ruta absoluta se realiza desde la raíz de la unidad de disco en la que se está trabajando y la relativa cuenta desde la carpeta actual de trabajo.

Otra posibilidad de construcción es utilizar como primer parámetro un objeto File ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.

```
File carpeta1=new File("c:/datos");//ó c\\datos
File archivo1=new File(carpeta1,"bd.txt");
```

Si el archivo o carpeta que se intenta examinar no existe, la clase *File* no devuelve una excepción. Habrá que utilizar el método **exists**. Este método recibe **true** si la carpeta o archivo es válido (puede provocar excepciones *SecurityException*).

También se puede construir un objeto **File** a partir de un objeto **URL**.

el problema de las rutas

Cuando se crean programas en Java hay que tener muy presente que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la realización de rutas sea problemática porque la forma de denominar y recorrer rutas es distinta en cada sistema operativo.

Por ejemplo en Windows se puede utilizar la barra / o la doble barra invertida \\ como separador de carpetas, en muchos sistemas Unix sólo es posible la primera opción. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta de elige desde un cuadro y, sobre todo, son independientes de la plataforma.

También se pueden utilizar las **variables estáticas** que posee File. Estas son:

propiedad	uso
char **separatorChar**	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es "/" y en Windows es "\", que se debe escribir como \\, ya que el carácter \ permite colocar caracteres de control, de ahí que haya que usar la doble barra.
String **separator**	Como el anterior pero en forma de String
char **pathSeparatorChar**	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser ":", en Windows es ";"
String **pathSeparator**	Como el anterior, pero en forma de String

Para poder garantizar que el separador usado es el del sistema en uso:

```
String ruta="documentos/manuales/2003/java.doc";
ruta=ruta.replace('/',File.separatorChar);
```

Normalmente no es necesaria esta comprobación ya que Windows acepta también el carácter / como separador.

métodos generales

método	uso
String **toString()**	Para obtener la cadena descriptiva del objeto
boolean **exists()**	Devuelve **true** si existe la carpeta o archivo.
boolean **canRead()**	Devuelve **true** si el archivo se puede leer
boolean **canWrite()**	Devuelve **true** si el archivo se puede escribir
boolean **isHidden()**	Devuelve **true** si el objeto File es oculto
boolean **isAbsolute()**	Devuelve **true** si la ruta indicada en el objeto File es absoluta
boolean **equals(File** f2**)**	Compara f2 con el objeto **File** y devuelve verdadero si son iguales.
String **getAbsolutePath()**	Devuelve una cadena con la ruta absoluta al objeto File.
File **getAbsoluteFile()**	Como la anterior pero el resultado es un objeto File
String **getName()**	Devuelve el nombre del objeto File.
String **getParent()**	Devuelve el nombre de su carpeta superior si la hay y si no **null**
File **getParentFile()**	Como la anterior pero la respuesta se obtiene en forma de objeto File.
boolean **setReadOnly()**	Activa el atributo de sólo lectura en la carpeta o archivo.

método	uso
URL **toURL()** throws MalformedURLException	Convierte el archivo a su notación URL correspondiente
URI **toURI()**	Convierte el archivo a su notación URI correspondiente

métodos de carpetas

método	uso
boolean **isDirectory()**	Devuelve **true** si el objeto File es una carpeta y **false** si es un archivo o si no existe.
boolean **mkdir()**	Intenta crear una carpeta y devuelve **true** si fue posible hacerlo
boolean **mkdirs()**	Usa el objeto para crear una carpeta con la ruta creada para el objeto y si hace falta crea toda la estructura de carpetas necesaria para crearla.
boolean **delete()**	Borra la carpeta y devuelve **true** si puedo hacerlo
String[] **list()**	Devuelve la lista de archivos de la carpeta representada en el objeto File.
static File[] **listRoots()**	Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
File[] **listfiles()**	Igual que la anterior, pero el resultado es un array de objetos File.

métodos de archivos

método	uso
boolean **isFile()**	Devuelve **true** si el objeto File es un archivo y **false** si es carpeta o si no existe.
boolean **renameTo(**File f2**)**	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve **true** si se pudo completar la operación.
boolean **delete()**	Borra el archivo y devuelve **true** si puedo hacerlo
long **length()**	Devuelve el tamaño del archivo en bytes
boolean **createNewFile()** Throws IOException	Crea un nuevo archivo basado en la ruta dada al objeto File. Hay que capturar la excepción *IOException* que ocurriría si hubo error crítico al crear el archivo. Devuelve **true** si se hizo la creación del archivo vacío y **false** si ya había otro archivo con ese nombre.

método	uso
static File createTempFile(String prefijo, **String** sufijo**)**	Crea un objeto File de tipo archivo temporal con el prefijo y sufijo indicados. Se creará en la carpeta de archivos temporales por defecto del sistema. El prefijo y el sufijo deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo **IllegalArgumentsException** Requiere capturar la excepción **IOException** que se produce ante cualquier fallo en la creación del archivo
static File createTempFile(String prefijo, **String** sufijo, **File** directorio**)**	Igual que el anterior, pero utiliza el directorio indicado.
void deleteOnExit()	Borra el archivo cuando finaliza la ejecución del programa

secuencias de archivo

lectura y escritura byte a byte

Para leer y escribir datos a archivos, Java utiliza dos clases especializadas que leen y escriben orientando a byte (Véase tema anterior); son **FileInputStream** (para la lectura) y **FileOutputStream** (para la escritura).

Se crean objetos de este tipo construyendo con un parámetro que puede ser una ruta o un objeto File:

```
FileInputStream fis=new FileInputStream(objetoFile);
FileInputStream fos=new FileInputStream("/textos/texto25.txt");
```

La construcción de objetos **FileOutputStream** se hace igual, pero además se puede indicar un segundo parámetro booleano que con valor **true** permite añadir más datos al archivo (normalmente al escribir se borra el contenido del archivo, valor **false**).

Estos constructores intentan abrir el archivo, generando una excepción del tipo **FileNotFoundException** si el archivo no existiera u ocurriera un error en la apertura. Los métodos de lectura y escritura de estas clases son los heredados de las clases InputStream y OutputStream. Los métodos **read** y **write** son los que permiten leer y escribir. El método **read** devuelve -1 en caso de llegar al final del archivo.

Otra posibilidad, más interesante, es utilizar las clases DataInputStream y DataOutputStream. Estas clases está mucho más preparadas para escribir datos de todo tipo.

escritura

El proceso sería:

1> Crear un objeto **FileOutputStream** a partir de un objeto **File** que posee la ruta al archivo que se desea escribir.

2> Crear un objeto **DataOutputStream** asociado al objeto anterior. Esto se realiza en la construcción de este objeto.

3> Usar el objeto del punto 2 para escribir los datos mediante los métodos **write*Tipo*** donde *tipo* es el tipo de datos a escribir (Int, Double, ...). A este método se le pasa como único argumento los datos a escribir.

4> Se cierra el archivo mediante el método **close** del objeto DataOutputStream**.**

Ejemplo:

```
File f=new File("D:/prueba.out");
Random r=new Random();
double d=18.76353;
try{
    FileOutputStream fis=new FileOutputStream(f);
    DataOutputStream dos=new DataOutputStream(fis);
    for (int i=0;i<234;i++){ //Se repite 233 veces
        dos.writeDouble(r.nextDouble());//Nº aleatorio
    }
    dos.close();
}
catch(FileNotFoundException e){
    System.out.println("No se encontro el archivo");
}
catch(IOException e){
    System.out.println("Error al escribir");
}
```

lectura

El proceso es análogo. Sólo que hay que tener en cuenta que al leer se puede alcanzar el final del archivo. Al llegar al final del archivo, se produce una excepción del tipo **EOFException** (que es subclase de **IOException**), por lo que habrá que controlarla.

Ejemplo, leer los números del ejemplo anterior :

```
boolean finArchivo=false;//Para bucle infinito
try{
    FileInputStream fis=new FileInputStream(f);
    DataInputStream dis=new DataInputStream(fis);
```

```
    while (!finArchivo){
        d=dis.readDouble();
        System.out.println(d);
    }

    dis.close();
}
catch(EOFException e){
    finArchivo=true;
}
catch(FileNotFoundException e){
    System.out.println("No se encontro el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
```

En este listado, obsérvese como el bucle **while** que da lugar a la lectura se ejecuta indefinidamente (no se pone como condición a secas **true** porque casi ningún compilador lo acepta), se saldrá de ese bucle cuando ocurra la excepción EOFException que indicará el fin de archivo.

Las clases DataStream son muy adecuadas para colocar datos binarios en los archivos.

lectura y escritura mediante caracteres

Como ocurría con la entrada estándar, se puede convertir un objeto **FileInputStream** o **FileOutputStream** a forma de **Reader** o **Writer** mediante las clases **InputStreamReader** y **OutputStreamWriter**.

Existen además dos clases que manejan caracteres en lugar de bytes (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader.**

La construcción de objetos del tipo FileReader se hace con un parámetro que puede ser un objeto File o un String que representarán a un determinado archivo.

La construcción de objetos FileWriter se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer **true**, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Para escribir se utiliza **write** que es un método void que recibe como parámetro lo que se desea escribir en formato int, String o array de caracteres. Para leer se utiliza el método read que devuelve un int y que puede recibir un array de caracteres en el que se almacenaría lo que se desea leer. Ambos métodos pueden provocar excepciones de tipo IOException.

Ejemplo:

```
File f=new File("D:/archivo.txt");
int x=34;
```

```
try{
    FileWriter fw=new FileWriter(f);
    fw.write(x);
    fw.close();

}
catch(IOException e){
    System.out.println("error");
    return;
}
//Lectura de los datos
try{
    FileReader fr=new FileReader(f);
    x=fr.read();
    fr.close();
}
catch(FileNotFoundException e){
    System.out.println("Error al abrir el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
System.out.println(x);
```

En el ejemplo anterior, primero se utiliza un *FileWrite* llamado *fw* que escribe un valor entero (aunque realmente sólo se escribe el valor carácter, es decir sólo valdrían valores hasta 32767). La función *close* se encarga de cerrar el archivo tras haber leído. La lectura se realiza de forma análoga.

Otra forma de escribir datos (imprescindible en el caso de escribir texto) es utilizar las clases **BufferedReader** y **BufferedWriter** vistas en el tema anterior. Su uso sería:

```
File f=new File("D:/texto.txt);
int x=105;
try{
    FileReader fr=new FileReader(f);
    BufferedReader br=new BufferedReader(fr);
    String s;
    do{
        s=br.readLine();
        System.out.println(s);
    }while(s!=null);
}
```

```
catch(FileNotFoundException e){
    System.out.println("Error al abrir el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
```

En este caso el listado permite leer un archivo de texto llamado **texto.txt**. El fin de archivo con la clase BufferedReader se detecta comparando con **null**, ya que en caso de que lo leído sea null, significará que hemos alcanzado el final del archivo. La gracia de usar esta clase está en el método **readLine** que agiliza enormemente la lectura.

RandomAccessFile

Esta clase permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento. Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último.

Esta es una clase primitiva que implementa los interfaces **DataInput** y **DataOutput** y sirve para leer y escribir datos.

La construcción requiere de una cadena que contenga una ruta válida a un archivo o de un archivo File. Hay un segundo parámetro obligatorio que se llama **modo**. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

Como ocurría en las clases anteriores, hay que capturar la excepción **FileNotFound** cuando se ejecuta el constructor.

```
File f=new File("D:/prueba.out");
RandomAccessFile archivo = new RandomAccessFile( f, "rw");
```

Los métodos fundamentales son:

- **seek(**long pos**).** Permite colocarse en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es el puntero de acceso que es la señal que marca la posición a leer o escribir.
- long **getFilePointer().** Posición actual del puntero de acceso
- long **length().** Devuelve el tamaño del archivo
- **readBoolean, readByte, readChar, readInt, readDouble, readFloat, readUTF, readLine.** Funciones de lectura. Leen un dato del tipo indicado. En el caso de *readUTF* lee una cadena en formato Unicode.
- **writeBoolean, writeByte, writeBytes, writeChar, writeChars writeInt, writeDouble, writeFloat, writeUTF, writeLine.** Funciones de escritura. Todas reciben como parámetro, el dato a escribir. Escribe encima de lo ya escrito. Para escribir al final hay que colocar el puntero de acceso al final del archivo.

el administrador de seguridad

Llamado **Security manager**, es el encargado de prohibir que subprogramas y aplicaciones escriban en cualquier lugar del sistema. Por eso numerosas acciones podrían dar lugar a excepciones del tipo **SecurityException** cuando no se permite escribir o leer en un determinado sitio.

serialización

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz **serializable** que es la que permite esta operación. Si un objeto ejecuta esta interfaz puede ser guardado y restaurado mediante una secuencia.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción **`implements`** `Serializable` (además de importar la clase **java.io.Serializable**) en la cabecera de clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que el objeto sea serializable.

La clase **ObjectInputStream** y la clase **ObjectOutputStream** se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de **InputStream** y **OutputStream**, de hecho son casi iguales a **DataInput/OutputStream** sólo que incorporan los métodos **readObject** y **writeObject** que son muy poderosos. Ejemplo:

```
try{
   FileInputStream fos=new FileInputStream("d:/nuevo.out");
  ObjectInputStream os=new ObjectInputStream(fos);
  Coche c;
  boolean finalArchivo=false;

  while(!finalArchivo){
        c=(Coche) readObject();
        System.out.println(c);
   }
}
catch(EOFException e){
   System.out.println("Se alcanzó el final");
}
catch(ClassNotFoundException e){
   System.out.println("Error el tipo de objeto no es compatible");
}
catch(FileNotFoundException e){
   System.out.println("No se encontró el archivo");
}
catch(IOException e){
   System.out.println("Error  "+e.getMessage());
   e.printStackTrace();
```

El listado anterior podría ser el código de lectura de un archivo que guarda coches. Los métodos **readObject** y **writeObject** usan objetos de tipo Object, readObject les devuelve y writeObject les recibe como parámetro. Ambos métodos lanzan excepciones del tipo **IOException** y **readObject** además lanza excepciones del tipo **ClassNotFoundException**.

Obsérvese en el ejemplo como la excepción **EOFException** ocurre cuando se alcanzó el final del archivo.

clases fundamentales (II)
colecciones

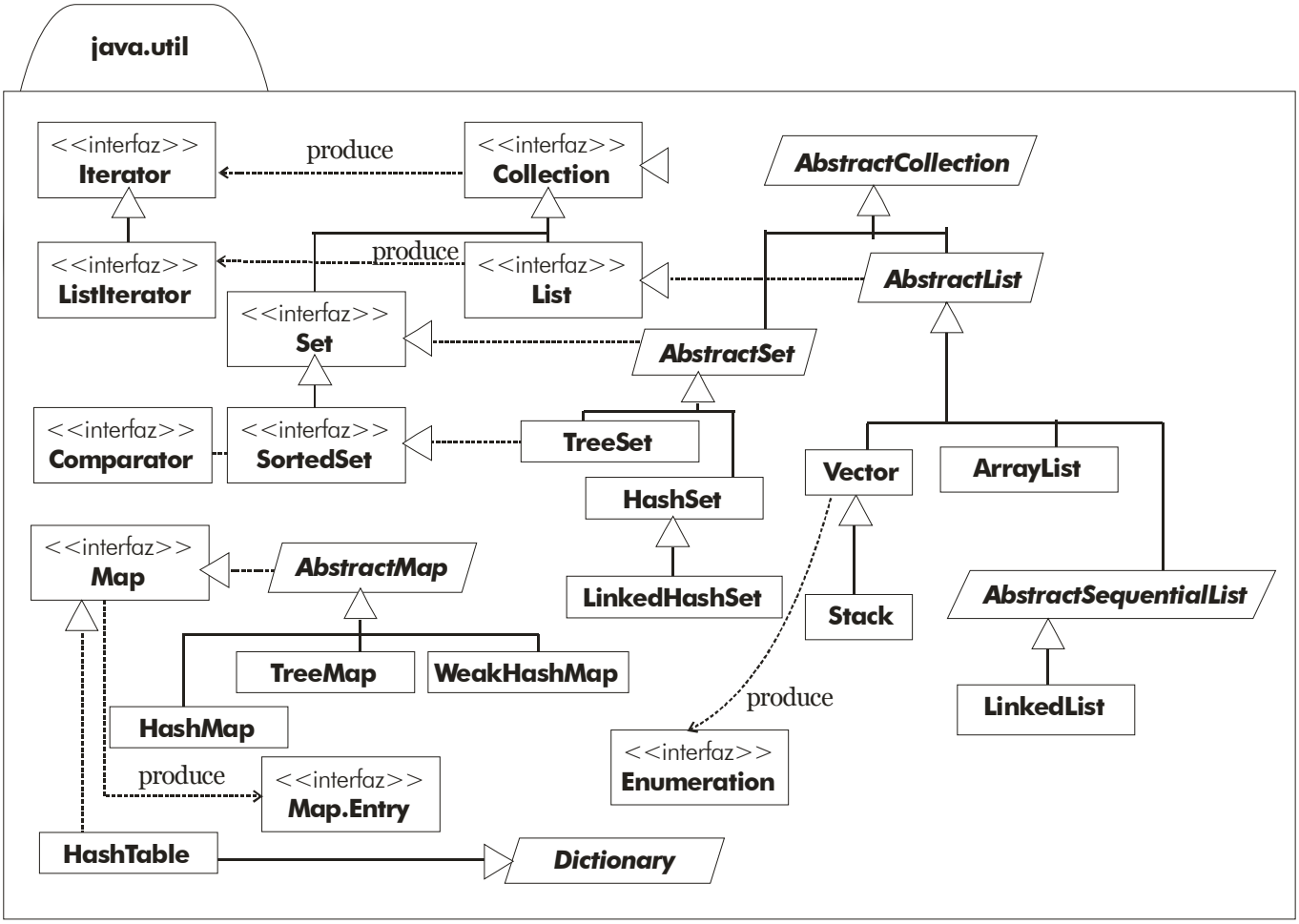


Ilustración 17, Diagrama de clases UML de las clases de colecciones

estructuras estáticas de datos y estructuras dinámicas

En prácticamente todos los lenguajes de computación existen estructuras para almacenar colecciones de datos. Esto es una serie de datos agrupados a los que se puede hacer referencia con un único nombre. Ejemplo de ello son los arrays (*véase* arrays, *página 31*). La pega de los arrays es que es una estructura estática, esto significa que se debe saber el número de elementos que formarán parte de esa colección a priori, es decir en tiempo de compilación hay que decidir el tamaño de un array.

Las estructuras dinámicas de datos tienen la ventaja de que sus integrantes se deciden en tiempo de ejecución y que el número de elementos es ilimitado. Estas estructuras dinámicas son clásicas en la programación y son las colas, pilas, listas enlazadas, árboles, grafos, etc. En muchos lenguajes se implementan mediante punteros; como Java no posee punteros se crearon clases especiales para implementar estas funciones.

En Java desde la primera versión se incluyeron las clases: **vector, Stack, Hashtable, BitSet** y la interfaz **Enumeration.** En Java 2 se modificó este funcionamiento y se potenció la creación de estas clases.

interfaz Collection

La interfaz fundamental de trabajo con estructuras dinámicas es **java.util.Collection**. Esta interfaz define métodos muy interesantes para trabajar con listas. Entre ellos:

método	uso
boolean add(Object o)	Añade el objeto a la colección. Devuelve **true** si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve **false**
boolean remove(Object o)	Elimina al objeto indicado de la colección.
int size()	Devuelve el número de objetos almacenados en la colección
boolean isEmpty()	Indica si la colección está vacía
boolean contains(Object o)	Devuelve **true** si la colección contiene a *o*
void clear()	Elimina todos los elementos de la colección
boolean addAll(Collection otra**)**	Añade todos los elementos de la colección *otra* a la colección actual
boolean removeAll(Collection otra**)**	Elimina todos los objetos de la colección actual que estén en la colección *otra*
boolean retainAll(Collection otra**)**	Elimina todos los elementos de la colección que no estén en la otra
boolean containsAll(Collection otra**)**	Indica si la colección contiene todos los elementos de otra
Object[] toArray()	Convierte la colección en un array de objetos.
Iterator iterator()	Obtiene el objeto iterador de la colección

iteradores

La interfaz **Iterator** (también en **java.util)** define objetos que permiten recorrer los elementos de una colección. Los métodos definidos por esta interfaz son:

método	uso
Object next()	Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RunTimeException**)
boolean hasNext()	Indica si hay un elemento siguiente (y así evita la excepción).
void remove()	Elimina el último elemento devuelto por **next**

Ejemplo (recorrido por una colección):

```
Iterator it=colecciónString.iterator();
while(it.hasNext()){
   String s=(String)it.next(); System.out.println(s);}
```

Listas enlazadas

interfaz *List*

List es una interfaz (de **java.util**) que se utiliza para definir listas enlazadas. Las listas enlazadas son colecciones de datos en las que importa la posición de los objetos. Deriva de la interfaz *Collection* por lo que hereda todos sus métodos. Pero los interesantes son los que aporta esta interfaz:

método	uso
void add(int índice, **Object** elemento**)**	Añade el elemento indicado en la posición *índice* de la lista
void remove(int índice**)**	Elimina el elemento cuya posición en la colección la da el parámetro *índice*
Object set(int índice**, Object** elemento**)**	Sustituye el elemento número *índice* por uno nuevo. Devuelve además el elemento antiguo
int indexOf(Object elemento**)**	Devuelve la posición del elemento. Si no lo encuentra, devuelve -1
int lastIndexOf(Object elemento**)**	Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1
void addAll(int índice, **Collection** elemento**)**	Añade todos los elementos de una colección a una posición dada.
ListIterator listIterator()	Obtiene el iterador de lista que permite recorrer los elementos de la lista
ListIterator listIterator(int índice**)**	Obtiene el iterador de lista que permite recorrer los elementos de la lista. El iterador se coloca inicialmente apuntando al elemento cuyo índice en la colección es el indicado.

ListIterator

Es un interfaz que define clases de objetos para recorrer listas. Es heredera de la interfaz **Iterator.** Aporta los siguientes métodos

método	uso
void add(Object elemento**)**	Añade el elemento delante de la posición actual del iterador
void set(Object *elemento***)**	Sustituye el elemento señalado por el iterador, por el elemento indicado

método	uso
Object **previous**()	Obtiene el elemento previo al actual. Si no lo hay provoca excepción: **NoSuchElementException**
boolean **hasPrevious**()	Indica si hay elemento anterior al actualmente señalado por el iterador
int **nextIndex()**	Obtiene el índice del elemento siguiente
int **previousIndex()**	Obtiene el índice del elemento anterior
List **subList(**int desde, int hasta**)**	Obtiene una lista con los elementos que van de la posición *desde* a la posición *hasta*

clase *ArrayList*

Implementa la interfaz *List*. Está pensada para crear listas en las cuales se aumenta el final de la lista frecuentemente. Disponible desde la versión 1.2

Posee tres constructores:

- **ArrayList().** Constructor por defecto. Simplemente crea un ArrayList vacío
- **ArrayList(**int capacidadInicial**).** Crea una lista con una capacidad inicial indicada.
- **ArrayList(**Collection c**).** Crea una lista a partir de los elementos de la colección indicada.

clase LinkedList

Crea listas de adición doble (desde el principio y el final). Implementa la interfaz *List*. Desde este clase es sencillo implantar estructuras en forma de pila o de cola. Añade los métodos:

método	uso
Object **getFirst()**	Obtiene el primer elemento de la lista
Object **getLast()**	Obtiene el último elemento de la lista
void **addFirst(**Object o**)**	Añade el objeto al principio de la lista
void **addLast(**Object o**)**	Añade el objeto al final de la lista
void **removeFirst()**	Borra el primer elemento
void **removeLast()**	Borra el último elemento

colecciones sin duplicados

interfaz Set

Define métodos para crear listas dinámicas de elementos sin duplicados. Deriva de *Collection*. Es el método **equals** el que se encarga de determinar si dos objetos son duplicados en la lista (habrá que redefinir este método para que funcione adecuadamente).

Posee los mismos métodos que la interfaz **Collection**. La diferencia está en el uso de duplicados.

clase *HashSet*

Implementa la interfaz anterior.

árboles. *SortedSet*

Un árbol es una colección ordenada de elementos. Al recorrer esta estructura, los datos aparecen automáticamente en el orden correcto. La adición de elementos es más lenta, pero su recorrido ordenado es mucho más eficiente.

La interfaz **SortedSet** es la encargada de definir esta estructura. Esta interfaz deriva de **Collection** y añade estos métodos:

método	uso
Object **first()**	Obtiene el primer elemento del árbol (el más pequeño)
Object **last()**	Obtiene el último elemento del árbol (el más grande)
SortedSet **headSet(**Object o**)**	Obtiene un SortedSet que contendrá todos los elementos menores que el objeto *o*.
SortedSet **tailSet(**Object o**)**	Obtiene un SortedSet que contendrá todos los elementos mayores que el objeto *o*.
SortedSet **subSet(**Object menor, Object mayor**)**	Obtiene un SortedSet que contendrá todos los elementos del árbol cuyos valores ordenados estén entre el menor y mayor objeto indicado
Comparator **comparator()**	Obtiene el objeto comparador de la lista

El resto de métodos son los de la interfaz Collection (sobre todo add y remove). La clase **TreeSet** implementa esta interfaz.

comparaciones

El problema es que los objetos tienen que poder ser comparados para determinar su orden en el árbol. Esto implica implementar la interfaz **Comparable** de Java (está en **java.lang**). Esta interfaz define el método **compareTo** que utiliza como argumento un objeto a comparar y que devuelve 0 si los objetos son iguales, 1 si el primero es mayor que el segundo y -1 en caso contrario.

Con lo cual los objetos a incluir en un TreeSet deben implementar Comparator y esto les obliga a redefinir el método compareTo (recordando que su argumento es de tipo ***Object*).**

Otra posibilidad es utilizar un objeto **Comparator**. Esta es otra interfaz que define el método **compare** al que se le pasan dos objetos. Su resultado es como el de *compareTo* (0 si son iguales, 1 si el primero es mayor y -1 si el segundo es mayor). Para que un árbol utilice este tipo de objetos se les pasa como argumentos en su creación.

mapas

Permiten definir colecciones de elementos que poseen pares de datos clave-valor. Esto se utiliza para localizar valores en función de la clave que poseen. Son muy interesantes y rápidos. Es la nueva implementación de tablas hash (ver tablas hash, más adelante). Métodos:

método	uso
Object get(Object clave**)**	Devuelve el objeto que posee la clave indicada
Object put(Object clave, **Object** valor**)**	Coloca el par clave-valor en el mapa. Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null**
Object remove(Object clave**)**	Elimina de la lista el valor asociado a esa clave.
boolean containsKey(Object clave**)**	Indica si el mapa posee la clave señalada
boolean containsValue(Object valor**)**	Indica si el mapa posee el valor señalado
void putAll(Map mapa**)**	Añade todo el mapa al mapa actual
Set keySet()	Obtiene un objeto *Set* creado a partir de las claves del mapa
Collection values()	Obtiene la colección de valores del mapa
Set entrySet()	Devuelve una lista formada por objetos **Map.Entry**

El objeto **Map.Entry** es interno a los objetos **Map** y representa un objeto de par clave/valor. Tiene estos métodos:

método	uso
Object getKey()	Obtiene la clave del elemento actual **Map.Entry**
Object getValue()	Obtiene el valor
Object setValue(Object valor**)**	Cambia el valor y devuelve el valor anterior del objeto actual

Esta interfaz está implementada en la clase **HashMap**. Además existe la interfaz **SortedMap** implementada en **TreeMap.** La diferencia es que *TreeMap* crea un árbol ordenado con las claves (el manejo es el mismo).

colecciones de la versión 1.0 y 1.1

En las primeras versiones de Java, el uso de las colecciones estaba restringido a las clases **Vector** y **Stack**. Desde Java 2 se anima a no utilizarlas, aunque se las mantiene por compatibilidad.

clase *Vector*

La clase **Vector** implementa la interfaz *List*. Es una clase veterana casi calcada a la clase *ArrayList*. En las primeras versiones de Java era la única posibilidad de implementar arrays dinámicos. Actualmente sólo se recomienda su uso si se utiliza una estructura dinámica para usar con varios *threads*.

Esto se debe a que esta clase implementa todos los métodos con la opción **synchronized**. Como esta opción hace que un método se ejecute más lentamente, se recomienda suplantar su uso por la clase *ArrayList* en los casos en los que la estructura dinámica no requiera ser sincronizada.

Otra diferencia es que permite utilizar la interfaz **Enumeration** para recorrer la lista de vectores. Las variables **Enumeration** tienen dos métodos **hasMoreElements** que indica si el vector posee más elementos y el método **nextElement** que devuelve el siguiente el elemento del vector (si no existiera da lugar a la excepción **NoSuchElementException**). La variable Enumeration de un vector se obtiene con el método **Elements** que devuelve una variable **Enumeration.**

clase Stack

Es una clase derivada de la anterior usada para crear estructuras de pilas. Las pilas son estructuras dinámicas en las que los elementos se añaden por arriba y se obtienen primero los últimos elementos añadidos. Sus métodos son:

método	uso
Object **push(**Object elemento**)**	Coloca un nuevo elemento en la pila
Object **pop()**	Retira el último elemento añadido a la pila. Si la pila está vacía, causa una excepción de tipo **EmptyStackException** (derivada de **RuntimeException**).
Object **peek()**	Obtiene el último elemento de la pila, pero sin retirarlo.

clase abstracta Dictionary

La desventaja de las estructuras anteriores reside en que su manipulación es larga debido a que el orden de la lista permanece en todo momento, lo que obliga a recorrer la lista elemento a elemento.

La clase abstracta Dictionary proporciona la interfaz para trabajar con mapas de valores clave. Actualmente está absolutamente reemplazado con la interfaz Map. La idea es proporcionar estructuras con un par de valores, código - contenido. En estas estructuras se busca el código para obtener el contenido.

tablas Hash

La clase **HashTable** implementa la clase anterior y proporciona métodos para manejar la estructura de los datos. Actualmente está en desuso ya que hay clases más poderosas.

En estas tablas cada objeto posee un código hash que es procesado rápidamente. Los elementos que tengan el mismo código hash, forman una lista enlazada. Con lo cual una tabla hash es una lista de listas enlazadas, asociadas con el mismo código.

Cada elemento a añadir en una tabla hash posee una clave y un valor. Ambos elementos son de tipo **Object**. La clave está pensada para ser buscada de forma rápida. La idea es que a partir de la clave obtenemos el objeto.

En las tablas hash es fundamental la obtención del código hash de cada objeto. Esto lo realiza el método **hashCode** que permite conocer el código hash de un objeto. Este método está implementado en la clase *Object*, pero a veces hay que redefinirlo en las clases de usuario para que funcione de manera conveniente. En cualquier caso, con los mismos datos, el algoritmo hashCode, obtiene el mismo código.

No obstante el método **hasCode** se puede redefinir para calcular el código de la forma que se estime conveniente.

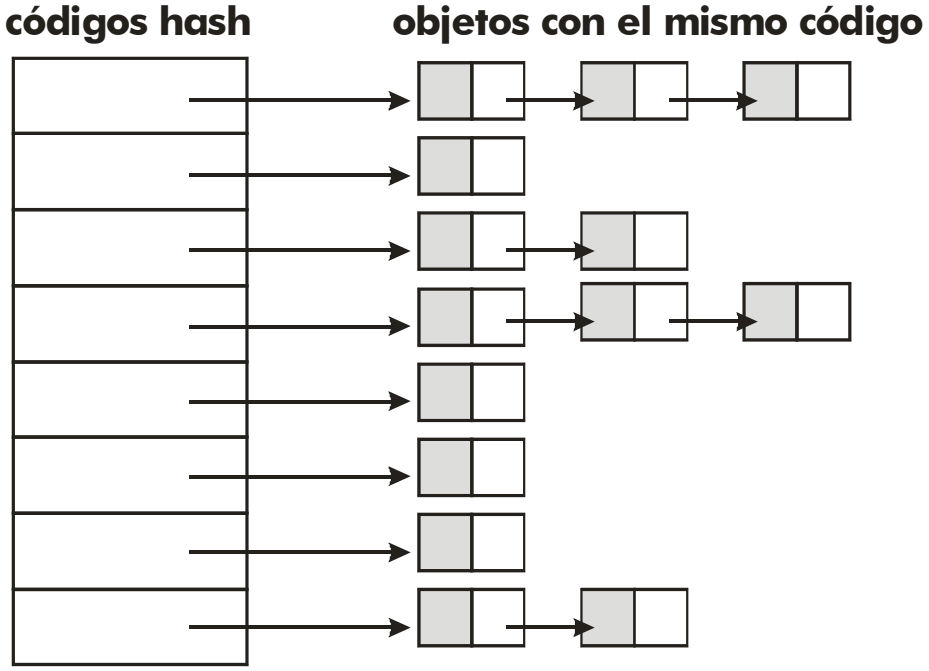


Ilustración 18, Estructura de las tablas hash

Métodos de **Hashtable**:

método	uso
Object **put(**Object clave, Object elemento**)**	Asocia la clave indicada al elemento. Devuelve excepción **NullPointerException** si la clave es nula. Devuelve el valor previo para esa misma clave
Object **get(**Object key)	Obtiene el valor asociado a esa clave.
Object **remove(**Object key)	Elimina el valor asociado a la clave en la tabla hash. Además devuelve ese valor
int **size()**	Obtiene el número de claves de la tabla hash
Enumeration **keys()**	Obtiene un objeto de enumeración para recorrer las claves de la tabla

método	uso
Enumeration **element()**	Obtiene un objeto de enumeración para recorrer los valores de la tabla
Set **keySet()**	(versión 1.2) Obtiene un objeto **Set** con los valores de las claves
Set **entrySet()**	(versión 1.2) Obtiene un objeto **Set** con los valores de la tabla
boolean **containsKey(**Object key)	**true** si la clave está en la tabla
boolean **containsValue(**Object valor)	**true** si el valor está en la tabla

la clase *Collections*

Hay una clase llamada **Collections** (no confundir con la interfaz **Collection**) que permite obtener fácilmente colecciones especiales, esto es lo que se conoce como envoltorio. Sus métodos son:

método	uso
static Collection **synchronizedCollection(**Collection c**)**	Obtiene una colección con métodos sincronizados a partir de la colección c
static List **synchronizedList(**List c**)**	Obtiene una lista con métodos sincronizados a partir de la lista c
static Set **synchronizedSet(**Set c**)**	Obtiene una tabla hash sincronizada a partir de la tabla hash c
static Set **synchronizedSortedSet(** SortedSet c**)**	Obtiene un árbol sincronizado a partir del árbol c
static Map **synchronizedMap(**Map c**)**	Obtiene un mapa sincronizado a partir del mapa c
static SortedMap **synchronizedSortedMap(**SortedMap c**)**	Obtiene un mapa ordenado sincronizado a partir de c
static Collection **unmodifiableCollection(**Collection c**)**	Obtiene una colección de sólo lectura a partir de la colección c
static List **unmodifiableList(**List c**)**	Obtiene una lista de sólo lectura a partir de la lista c
static Set **unmodifiableSet(**Set c**)**	Obtiene una tabla hash de sólo lectura a partir de la tabla hash c
static Set **unmodifiableSortedSet(** SortedSet c**)**	Obtiene un árbol de sólo lectura a partir de el árbol c
static Map **unmodifiableMap(**Map c**)**	Obtiene un mapa de sólo lectura a partir del mapa c
static SortedMap **unmodifiableSortedMap(**SortedMap c**)**	Obtiene un mapa ordenado de sólo lectura a partir de c
static void **sort(**List l**)**	Ordena la lista
static void **sort(**List l, Comparator c**)**	Ordena la lista basándose en el comparador indicado

método	uso
static int binarySearch(List l, **Object** o**)**	Busca de forma binaria el objeto en la lista (la lista tiene que estar ordenada en ascendente)
static int binarySearch(List l, **Object** o, **Comparator** *c***)**	Busca de forma binaria el objeto en la lista. La lista tiene que estar ordenada en ascendente usando el objeto comparador *c*

clases fundamentales (y III)

números grandes

Cuando se manipulan números sobre los que se requiere gran precisión, los tipos estándar de Java (**int, long, double,** etc.) se quedan cortos. Por ello en el paquete **java.math** disponemos de dos clases dedicadas a la precisión de números.

clase BigInteger

Se utiliza para cuando se desean almacenar números que sobrepasan los 64 bits del tipo **long**.

creación

constructor	uso
BigInteger(String texto**)** **throws** NumberFormatException	Crea un objeto para enteros grandes usando el número representado por el texto. En el caso de que el número no sea válido se lanza una excepción **NumberFormatException**
BigInteger(String texto, int base**)** **throws** NumberFormatException	Constructor idéntico al anterior, excepto en que se utiliza una base numérica determinada por el parámetro *base*.
BigInteger(int tamaño, Random r**)**	Genera un número entero largo aleatorio. El número aleatorio abarca el tamaño indicado por el primer parámetro, que se refiere al número de bits que ocupará el número.

Otra forma de crear es mediante el método estático **valueOf** al cual se le puede pasar un entero **long** a partir del cual se devuelve un **BigInteger.** Ejemplo:

```
BigInteger bi=BigInteger.valueOf(2500);
```

métodos

método	uso
BigInteger **abs()**	Obtiene el valor absoluto del número.
BigInteger **add(**BigInteger entero**)**	Devuelve el resultado de sumar el entero actual con el pasado como parámetro
int **bitCount()**	Devuelve el número de bits necesarios para representar el número.
int **compareTo(**BigInteger entero**)**	Compara el entero actual con el utilizado como parámetro. Devuelve -1 si el segundo es mayor que el primero, 0 si son iguales y 1 si el primero era mayor.
BigInteger **divide(**BigInteger entero**)**	Devuelve el resultado de dividir el entero actual entre el parámetro

método	uso
double **doubleValue()**	Obtiene el valor del entero en forma de número **double.**
boolean **equals(**Object o)	Compara el objeto *o* con el entero actual y devuelve **true** si son iguales
double **floatValue()**	Obtiene el valor del entero en forma de número **float.**
BigDecimal **max(**BigDecimal decimal**)**	Devuelve el mayor de los dos números
BigDecimal **min(**BigDecimal decimal**)**	Devuelve el menor de los dos números
BigInteger **mod(**BigInteger entero**)**	Devuelve el resto que se obtiene de dividir el número actual entre el que se pasa como parámetro
BigInteger **multiply(**BigInteger entero**)**	Multiplica los dos números y devuelve el resultado.
BigInteger **negate()**	Devuelve el número multiplicado por menos uno.
BigInteger **probablePrime(**int bits, Random r**)**	Calcula un número primo cuyo tamaño en bits es el indicado y que es generado a partir el objeto aleatorio *r*. La probabilidad de que el número no sea primo es de 2^{-100}
BigInteger **subtract(**BigInteger entero**)**	Resta el entero actual menos el que se recibe como par
BigInteger **toBigInteger()**	Convierte el decimal en **BigInteger**
String **toString()**	Obtiene el número en forma de cadena
String **toString(**int radio**)**	Obtiene el número en forma de cadena usando la base indicada

clase BigDecimal

Se utiliza con más frecuencia, representa números reales de gran precisión. Se usa una escala de forma que el valor de la misma indica la precisión de los decimales. El redondeo se realiza a través de una de estas constantes:

constante	descripción
static int **ROUND_CEILING**	Redondea hacia el infinito positivo
static int **ROUND_DOWN**	Redondea hacia el número 0
static int **ROUND_FLOOR**	Hacia el infinito negativo
static int **ROUND_HALF_DOWN**	Redondea hacia el valor del dígito conexo o cero si coinciden ambos
static int **ROUND_HALF_EVEN**	Redondea hacia el valor del dígito conexo o a un número par si coinciden
static int **ROUND_HALF_UP**	Redondea hacia el valor del dígito conexo o se alejan del cero si coinciden
static int **ROUND_UNNECESSARY**	Se presentan los valores sin redondeos

constante	descripción
static int ROUND_UP	Se redondea alejándose del cero

constructores

constructor	uso
BigDecimal(BigInteger enteroGrande**)**	Crea un **BigDecimal** a partir de un **BigInteger**
BigDecimal(BigInteger enteroGrande, int escala**)**	Crea un **BigDecimal** a partir de un **BigInteger** y coloca la escala indicada (la escala determina la precisión de los decimales)
BigDecimal(String texto**)** **throws** NumberFormatException	Crea un objeto para decimales grandes usando el número representado por el texto. En el caso de que el número no sea válido se lanza una excepción **NumberFormatException** El formato del número suele estar en notación científica (2.3e+21)
BigDecimal (double n**)**	Crea un decimal grande a partir del número doble *n*

métodos

método	uso
BigDecimal **abs()**	Obtiene el valor absoluto del número.
BigDecimal **add(**BigDecimal decimal**)**	Devuelve el resultado de sumar el decimal actual con el pasado como parámetro
int **bitCount()**	Devuelve el número de bits necesarios para representar el número.
int **compareTo(**BigDecimal decimal**)**	Compara el decimal actual con el utilizado como parámetro. Devuelve -1 si el segundo es mayor que el primero, 0 si son iguales y 1 si el primero era mayor.
BigDecimal **divide(**BigDecimal decimal, int redondeo**)**	Devuelve el resultado de dividir el decimal actual entre el *decimal* usado como parámetro. Se le indica el modo de redondeo que es una de las constantes descritas anteriormente.
BigDecimal **divide(**BigDecimal decimal, **int** escala, int redondeo**)**	Idéntica a la anterior, sólo que ahora permite utilizar una escala concreta.
double **doubleValue()**	Obtiene el valor del decimal en forma de número **double.**
double **floatValue()**	Obtiene el valor del decimal en forma de número **float.**
boolean **equals(**Object o)	Compara el objeto *o* con el entero actual y devuelve **true** si son iguales

método	uso
BigDecimal **max(**BigDecimal decimal**)**	Devuelve el mayor de los dos números
BigDecimal **min(**BigDecimal decimal**)**	Devuelve el menor de los dos números
BigDecimal **multiply(**BigDecimal decimal**)**	Multiplica los dos números y devuelve el resultado.
BigDecimal **negate()**	Devuelve el número multiplicado por menos uno.
BigDecimal **subtract(**BigDecimal decimal**)**	Resta el entero actual menos el que se recibe como par
int **scale()**	Obtiene la escala del número
void **setScale(**int escala**)**	Modifica la escala del número
String **toString()**	Obtiene el número en forma de cadena

internacionalización. clase Locale

Una de las premisas de Java es la posibilidad de construir aplicaciones que se ejecuten en cualquier idioma y país.

la clase Locale

Toda la cuestión de la portabilidad del código a diversas lenguas, gira en torno a la clase **Locale** (en el paquete **java.util**)**.** Un objeto Locale identifica una configuración regional (código de país y código de idioma). En muchos casos los idiomas soportados están definidos mediante una serie de constantes, que son:

constante	significado
static Locale **CANADA**	**País Canadá**
static Locale **CANADA_FRENCH**	**Idioma francés de Canadá**
static Locale **CHINA**	**País China**
static Locale **CHINESE**	**Idioma chino**
static Locale **ENGLISH**	**Idioma inglés**
static Locale **FRANCE**	**País Francia**
static Locale **FRENCH**	**Idioma francés**
static Locale **GERMAN**	**Idioma alemán**
static Locale **GERMANY**	**País Alemania**
static Locale **ITALIAN**	**Idioma italiano**
static Locale **ITALY**	**País Italia**
static Locale **JAPAN**	**País Japón**
static Locale **JAPANESH**	**Idioma japonés**
static Locale **KOREA**	**País corea del sur**
static Locale **KOREAN**	**Idioma coreano**
static Locale **SIMPLIFIED_CHINESE**	**Idioma chino simplificado**
static Locale **TAIWAN**	**País Taiwán**
static Locale **TRADITIONAL_CHINESE**	**Idioma chino tradicional**

constante	significado
static Locale **UK**	**País Reino Unido**
static Locale **US**	**País Estados Unidos**

La creación de un objeto local para Italia sería:

```
Locale l=Locale.ITALY;
```

No obstante se puede crear cualquier configuración local mediante los constructores que son:

constructor	uso
Locale(String códigoLenguaje**)**	Crea un objeto **Locale** utilizando el código de lenguaje indicado
Locale(String códigoLenguaje, String códigoPaís)	Crea un objeto **Locale** utilizando los códigos de lenguaje y país indicados indicado

Los códigos de países se pueden obtener de la dirección http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt y los códigos de países de: http://ftp.ics.uci.edu/pub/ietf/http/related/iso3166.txt

métodos de Locale

método	descripción
String **getCountry()**	Escribe el código del país del objeto Locale (ES para España)
String **getDisplayCountry()**	Escribe el nombre del país del objeto Locale (España)
String **getLanguage()**	Escribe el código del idioma del objeto Locale (es para Español)
String **getDisplayLanguage()**	Escribe el nombre del idioma (español)
String **getDisplayName()**	Obtiene el texto completo de descripción del objeto local (español España)

métodos estáticos

Permiten obtener información sobre la configuración local de la máquina virtual de Java en ejecución.

método	descripción
static Locale[] **getAvailableLocales()**	Devuelve un array de objetos Locales con la lista completa de objetos locales disponibles en la máquina virtual en ejecución.
static Locale **getDefaultLocale()**	Obtiene el objeto local que se está utilizando en la máquina actual

método	descripción
static String[] getISOCountries()	Obtiene una lista con los códigos de países de dos letras disponibles en la máquina actual
static String[] getISOLanguages	Obtiene una lista con los códigos de idiomas letras disponibles en la máquina actual
static void setDefault(Locale l)	Modifica el objeto Locale por defecto de la máquina actual

formatos numéricos

Ya se vio anteriormente la clase **DateFormat** (ver **clase DateFormat**, página 90). En el mismo paquete de ésta (**java.text**) existen clases dedicadas también al formato de números. Para ello disponemos de **NumberFormat.**

Los objetos **NumberFormat** sirven para formatear números, a fin de mostrarles de forma conveniente. cada objeto **NumberFormat** representa un formato numérico.

creación

Hay tres formas de crear formatos numéricos. Las tres formas se realizan con métodos estáticos (al estilo de **DateFormat**):

método	descripción
static NumberFormat getInstance()	Obtiene un objeto de formato numérico según las preferencias locales actuales
static NumberFormat getInstance(Locale local**)**	Obtiene un objeto de formato numérico basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getCurrencyInstance()	Obtiene un objeto de formato de moneda según las preferencias locales actuales
static NumberFormat geCurrencyInstance(Locale local**)**	Obtiene un objeto de formato de moneda basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getPercentInstance()	Obtiene un objeto de formato porcentaje basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getPercentInstance(Locale local**)**	Obtiene un objeto de formato en porcentaje (0,2 se escribiría 20%) basado en las preferencias del objeto *local* indicado como parámetro

USO

El método **format** devuelve una cadena que utiliza el formato del objeto para mostrar el número que recibe como parámetro. Ejemplo:

```
NumberFormat nf=NumberFormat.getCurrencyInstance();
System.out.println(nf.format(1248.32));
//sale 1.248,32 €
```

métodos

Hay métodos que permiten variar el resultado del objeto **NumberFormat:**

método	descripción
boolean getGroupingUsed()	Indica si se está utilizando el separador de miles
int getMinimumFractionDigit()	Devuelve el número mínimo de decimales con el que se formateará a los números
int getMaximumFractionDigit()	Devuelve el número máximo de decimales con el que se formateará a los números
int getMinimumIntegerDigit()	Devuelve el número mínimo de números enteros (los que están ala izquierda del decimal) con el que se formateará a los números. Si hay menos números que el mínimo, se rellenarán los que falta con ceros a la izquierda
int getMaximumIntegerDigit()	Devuelve el número máximo de números enteros con el que se formateará a los números
void setGroupingUsed(boolean uso**)**	Modifica el hecho de que se muestren o no los separadores de miles. Con valor **true** se mostrarán
void setMinimumFractionDigit(int n**)**	Establece el número mínimo de decimales
void setMaximumFractionDigit(int n**)**	Establece el número máximo de decimales
void setMinimumIntegerDigit(int n**)**	Establece el número mínimo de enteros
void setMaximumIntegerDigit(int n**)**	Establece el número máximo de enteros

Propiedades

Las propiedades permiten cargar valores de entorno, esto es valores que se utilizan durante la ejecución de los programas, pero que se almacenan de modo independiente a estos.

Es la clase **Properties (java.util)** la encargada de implementar las propiedades. Esta clase deriva de **Hashtable** con lo que los valores se almacenan en una lista de tipo clave / valor. En el caso de Properties la lista se construye con pares nombre / valor; donde el nombre es el nombre de la propiedad (que se puede agrupar mediante puntos, como los paquetes de Java).

USO

El método para colocar elementos en la lista de propiedades es **put**. Desde la versión 1.2 se utiliza **setProperty** (equivalente a put, pero más coherente con el método de obtención de propiedades **getProperty).** Este método usa dos cadenas, la primera es el nombre de la propiedad y el segundo su valor:

```
Properties prop1=new Properties();
```

```
prop1.setProperty("MiPrograma.maxResult","134");
prop1.setProperty("MiPrograma.minResult","5");
```

Para leer los valores se usa **getProperty,** que devuelve el valor del nombre de propiedad buscado o **null** si no lo encuentra.

```
String mResult=prop1.getProperty("MiPrograma.maxResult");
```

grabar en disco

La ventaja de la clase **Properties** es que las tablas de propiedades se pueden almacenar en discos mediante el método **store**. El método store (que sustituye al obsoleto, **save**) posee dos parámetros, el primero es un objeto de tipo **OutputStream** (ver InputStream/ OutputStream página 93) referido a una corriente de datos de salida en la que se grabará (en forma de texto ASCII) el contenido de la tabla de propiedades. El segundo parámetro es la cabecera de ese archivo. Por ejemplo:

```
prop1.save(System.out,"parámetros de programa");
try{
   File f=new File("d:/propiedades.out");
   FileOutputStream fos=new FileOutputStream(f);
   prop1,save(fos,"parámetros de programa");
}
catch(FileNotFoundException fnf){
   System.out.println("No se encontró el archivo");
}
/*en ambos casos la salida sería:
#parámetros de programa
#Wed Apr 28 00:01:30 CEST 2004
MiPrograma.maxResult=134
MiPrograma.minResult=5
*/
```

A su vez, el método **load** permite leer un archivo de propiedades. Este método devuelve un objeto **Properties** a partir de una corriente **InputStream.**

En ambos métodos (**load** y **store**) hay que capturar las posibles excepciones de tipo **IOException** que se produjeran.

propiedades del sistema

El objeto **System** que se encuentra en **java.lang** representa al propio sistema. Posee un método llamado **getProperty** que permite extraer sus propiedades (el objeto **System** posee una tabla preconfigurada de propiedades). Estas propiedades ocupan el lugar de las variables de entorno de los sistemas operativos.

Las posibles propiedades de sistema que se pueden extraer son:

propiedad	descripción
java.vendor	Fabricante de la máquina Java en ejecución
java.vendor.url	Dirección de la web del fabricante
java.version	Versión de java en ejecución
java.home	Directorio de instalación del kit de java en la máquina actual (inaccesible desde un applet)
os.arch	Arquitectura de Sistema Operativo
os.version	Versión de sistema operativo
os.name	Nombre del sistema operativo
file.separator	Separador de carpetas (\ en Windows)
path.separator	Separador de rutas (; en Windows)
line.separator	Separador de líneas (por ejemplo \n)
user.name	Nombre de usuario (inaccesible desde un applet)
user.home	Ruta a la carpeta de usuario (inaccesible desde un applet)
user.dir	Carpeta actual de trabajo (inaccesible desde un applet)
user.language	Código del lenguaje que utiliza el usuario (por ejemplo "es")
user.country	Código del país del usuario (por ejemplo "ES")
user.timezone	Código de la zona horaria del usuario (por ejemplo *Europe/Paris*)

temporizador

Desde la versión 1.3 de Java hay dos clases que permiten trabajar con temporizadores. Son **java.util.Timer** y **java.util.TimerTask.**

clase TimerTask

Representa una tarea de temporizador; es decir una tarea que se asignará a un determinado temporizador.

Se trata de una clase abstracta, por lo que hay que definir descendientes para poder utilizarla. Cuando se redefine una subclase se debe definir el método abstracto **run**. Este método es el que realiza la operación que luego se asignará a un temporizador.

El método **cancel** permite cancelar la ejecución de la tarea. Si la tarea ya había sido ejecutada, devuelve **false.**

clase Timer

Es la que representa al temporizador. El temporizador se crea usando el constructor por defecto. Mediante el método **schedule** se consigue programar el temporizador. Este

método tiene como primer parámetro un objeto **TimerTask** que tiene que haber programado la acción a realizar cuando se cumpla el tiempo del temporizador.

El segundo parámetro de **schedule** es un objeto **Date** que sirve para indicar cuándo se ejecutará el temporizador.

```
TimerTask tarea=new TimerTask(){
        public void run() {
              System.out.println("Felicidades!!");
        }
};
Timer temp=new Timer();
GregorianCalendar gc=new GregorianCalendar(2007,1,1);
temp.schedule(tarea,gc.getTime());
```

En el ejemplo anterior se escribirá Felicidades!!!, cuando estemos a 1 de enero de 2007 (o en cualquier fecha posterior).

Se puede añadir a **schedule** un tercer parámetro que permite especificar una repetición mediante un número **long** que indica milisegundos:

```
TimerTask tarea=new TimerTask(){
        public void run() {
              System.out.println("Felicidades!!");
        }
};
Timer temp=new Timer();
temp.schedule(tarea,new Date(), 1000);
```

En este caso se ejecuta la tarea en cada segundo. El segundo parámetro en lugar de ser un objeto Date() puede ser también un número de milisegundos desde el momento actual. Así la última línea podía hacerse también con:

```
temp.schedule(tarea, 0, 1000);
```

Finalmente añadir que el método **cancel**, que no tiene argumentos, permite finalizar el temporizador actual.

Swing

AWT y Swing

Swing es un conjunto de clases desarrolladas por primera vez para Java 1.2 (el llamado Java2), para mejorar el anterior paquete que implementaba clases para fabricar interfaces de usuario, el llamado AWT (*Abstract Window Tools*) que aún se usa bastante en las aplicaciones Java.

Tanto Swing como AWT forman parte de una colección de clases llamada **JFC** (*Java Foundation Classes*) que incluyen paquetes dedicados a la programación de interfaces gráficos (así como a la producción multimedia).

Uno de los problemas frecuentes de la programación clásica era como programar interfaces de usuario, ya que esto implicaba tener que utilizar las API propias del Sistema Operativo y esto provocaba que el código no fuera transportable a otros sistemas.

AWT fue la primera solución a este problema propuesta por Java. AWT está formada por un conjunto de clases que no dependen del sistema operativo, pero que proponen una serie de clases para la programación de GUIs (*graphic users interfaces,* interfaces gráficos de usuario; cualquier entorno de comunicación entre el ordenador y el usuario).

AWT usa clases gráficas comunes a todos los sistemas operativos gráficos y luego la máquina virtual traduce esa clase a la forma que tenga en el sistema concreto en el que se ejecutó el programa, sin importar que dicho sistema sea un sistema X, McIntosh o Windows. La popularidad de AWT desbordó las expectativas de la propia empresa Sun.

La clave de AWT era el uso de componentes *iguales* (*peers*). Los elementos de los interfaces AWT dejaban al sistema la responsabilidad de generar realmente los componentes. Eso aseguraba una vista coherente respecto al sistema en el que se ejecutaba el programa. El problema es que ante la grandiosidad de la imagen en Windows y Mac OS, otros sistemas quedaban peor ante la misma aplicación.

Por ello (y por otros problemas) aparece Swing en la versión 1.2 como parte del JFC (*Java Foundation Classes*) que es el kit de clases más importante de Java para las producciones gráficas.

Los problemas de AWT eran:

- AWT tenía problemas de compatibilidad en varios sistemas.
- A AWT le faltaban algunos componentes avanzados (árboles, tablas,...).
- Consumía excesivos recursos del sistema.

Swing aporta muchas más clases, consume menos recursos y construye mejor la apariencia de los programas. En cualquier caso, AWT no desaparece; simplemente se añade a las nuevas capacidades Swing

componentes

Los componentes son los elementos básicos de la programación con Swing. Todo lo que se ve en un GUI de Java es un componente. Los componentes se colocan en otros elementos llamados **contenedores** que sirven para agrupar componentes. Un

administrador de diseño se encarga de de disponer la presentación de los componentes en un dispositivo de presentación concreto.

La clase **javax.swing.JComponent** es la clase padre de todos los componentes. A su vez, **JComponent** desciende de **java.awt.container** y ésta de **java.awt.component.** De esto se deduce que Swing es una extensión de AWT, de hecho su estructura es análoga.

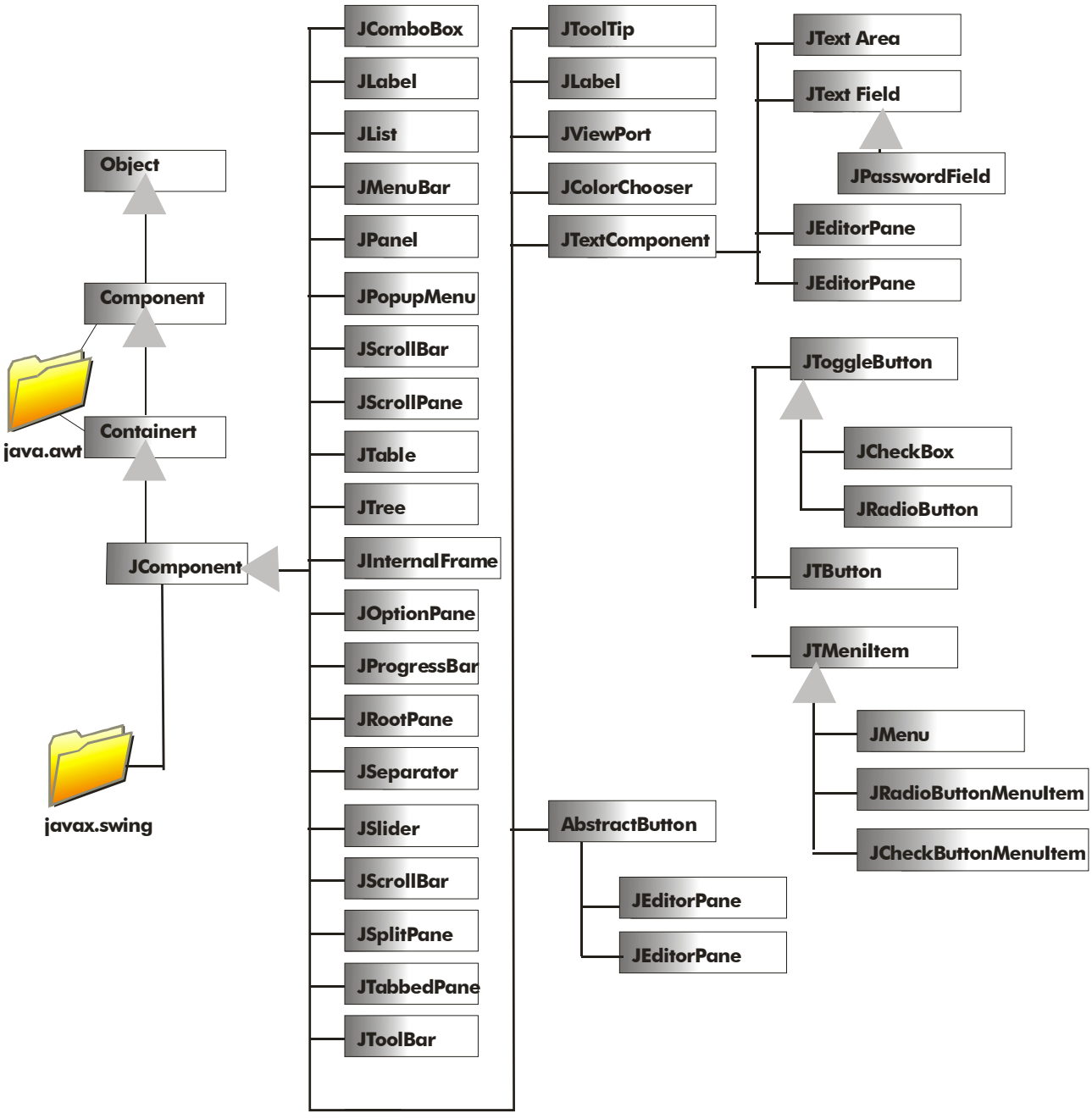


Ilustración 19, Referencia de las clases Swing

La clase JComponent posee métodos para controlar la apariencia del objeto. Por ejemplo: la visibilidad, tamaño, posición, tipo de letra, color,... Al dibujar un componente, se le asigna un **dispositivo de presentación.**

Además posee métodos que controlan el comportamiento del componente. Cuando el usuario ejecuta una acción sobre un componente, entonces se crea un objeto de **evento** que describe el suceso. El objeto de evento se envía a objetos de control de eventos (***Listeners***). Los eventos son uno de los pilares de la construcción de Interfaces de usuario y una de las bases de la comunicación entre objetos.

pares

En AWT se usaban interfaces de ***pares***. Esto significaba que cada componente creado con AWT, creaba un par igual correspondiente al mundo real. Es decir al crear el botón, existía el botón virtual creado en Java y el que realmente era dibujado en la pantalla (el real). El programador no necesita saber de la existencia de ese par, la comunicación del objeto creado con su par corría por cuenta de AWT.

Este modelo de componentes se elimina en Swing. En Swing se habla de componentes de *peso ligero*. La clase *JComponent* que es la raíz de clases Swing, no utiliza un par, cada componente es independiente del sistema de ventanas principal. Se dibujan a sí mismos y responden a los eventos de usuario sin ayuda de un par.

La ventaja de este modelo es que requiere menos recursos y que su modificación visual es más ágil y efectiva.

modelo/vista/controlador

Se trata del modelo fundamental del trabajo con interfaces de usuario por parte de Swing. Consiste en tres formas de abstracción. Un mismo objeto se ve de esas tres formas:

- **Modelo.** Se refiere al modelo de datos que utiliza el objeto. Es la información que se manipula mediante el objeto Swing.
- **Vista.** Es cómo se muestra el objeto en la pantalla.
- **Controlador**. Es lo que define el comportamiento del objeto.

Por ejemplo un array de cadenas que contenga los meses del año, podría ser el **modelo** de un cuadro combinado de Windows. Un cuadro combinado es un rectángulo con un botón con una flecha que permite elegir una opción de una lista. La **vista** de ese cuadro es el hecho de mostrar esas cadenas en ese rectángulo con flecha. Y el **controlador** es la capa software que permite capturar el clic del ratón cuando apunta a la flecha del control a fin de mostrar y seleccionar el contenido.

métodos de JComponent

La clase JComponent es abstracta, lo cual significa que no puede crear objetos, pero sí es la superclase de todos los componentes visuales (botones, listas, paneles, applets,...) y por ello la lista de métodos es interminable, ya que proporciona la funcionalidad de todos los componentes. Además puesto que deriva de **Component** y **Container** tiene los métodos de estos, por ello aún es más grande esta lista. Algunos son:

métodos de información

método	uso
String getName()	Obtiene el nombre del componente
void setname(String nombre**)**	cambia el nombre del componente
Container getParent()	Devuelve el contenedor que sostiene a este componente

métodos de apariencia y posición

método	uso
void setVisible(boolean vis**)**	Muestra u oculta el componente según el valor del argumento sea **true** o **false**
Color getForeground()	Devuelve el color de frente en forma de objeto Color
void setForeGround(Color color**)**	Cambia el color frontal
Color getBackground()	Devuelve el color de fondo en forma de objeto *java.awt.Color*
void setBackground(Color color**)**	Cambia el color de fondo
Point getLocation()	Devuelve la posición del componente en forma de objeto Point
void setLocation(int x, **int** y**)**	Coloca el componente en la posición x, y
void setLocation(Point p**)**	Coloca el componente en la posición marcada por las coordenadas del punto P
Dimension getSize()	Devuelve el tamaño del componente en un objeto de tipo *java.awt.Dimension*.
void setSize(Dimension d**)**	
void setSize(int ancho, **int** alto**)**	Cambia las dimensiones del objeto en base a un objeto *Dimension* o indicando la anchura y la altura con dos enteros.
void setBounds(int x, **int** y, **int** ancho, **int** alto)	Determina la posición de la ventana (en la coordenada x, y) así como su tamaño con los parámetros *ancho* y *alto*
void setPreferredSize(Dimension d**)**	Cambia el tamaño preferido del componente. Este tamaño es el que el componente realmente quiere tener.
void setToolTipText(String texto**)**	Hace que el texto indicado aparezca cuando el usuario posa el cursor del ratón sobre el componente
String getToolTipText()	Obtiene el texto de ayuda del componente
Cursor getCursor()	Obtiene el cursor del componente en forma de objeto *java.awt.Cursor*
void setCursor(Cursor cursor**)**	Cambia el cursor del componente por el especificado en el parámetro.
void setFont(Font fuente**)**	Permite especificar el tipo de letra de la fuente del texto

método	uso
void putClientProperty(Object clave, **Object** valor**)**	

Los objetos **java.awt.Point** tienen como propiedades públicas las coordenadas **x** e **y.** Se construyen indicando el valor de esas coordenadas y disponen de varios métodos que permiten modificar el punto.

Los objetos **java.awt.Dimension** tienen como propiedades la propiedad **width** (anchura) y **height** (altura). El método **getDimension()** obtiene un objeto **Dimension** con los valores del actual y **setDimension()** es un método que permite cambiar las dimensiones de varias formas.

Los objetos **java.awt.Color** representan colores y se pueden construir de las siguientes formas:

- **Color(int** rojo, **int** verde, **int** azul**).** Construye un objeto color indicando los niveles de rojo, verde y azul.
- **Color(int** rgb**).** Crea un color usando un único entero que indica los niveles de rojo, verde y azul. Se suele emplear con 6 dígitos en hexadecimal. Ejemplo: 0xFFCC33
- **Color(int** rojo, **int** verde, **int** azul, **int** alfa**).** Construye un objeto color indicando los niveles de rojo, verde y azul, y un valor de 0 a 255 indicando el valor alfa (alfa indica la transparencia).
- **Color(int** rgb**).** Crea un color usando un único entero que indica los niveles de rojo, verde y azul. Se suele emplear con 6 dígitos en hexadecimal. Ejemplo: 0xFFCC33
- **Color(int** rgb, **int** alfa**).**

Además existen constantes de colores ya fabricados: **Color.RED, Color.YELLOW, Color.GREEN, Color.BLUE, Color.PINK, Color.GRAY, Color.CYAN, Color.DARK_GRAY, Color.LIGHT_GRAY, Color.MAGENTA, Color.PINK** y **Color.WHITE.** Los objetos Color poseen además métodos interesantes para manipular colores.

Finalmente **java.awt.Cursor** es una clase que representa cursores y que se crea indicando un número que se puede reemplazar por una serie de constantes estáticas de la propia clase Cursor, que representan cursores. Las constantes son: **Cursor.HAND_CURSOR, Cursor.WAIT_CURSOR,**

Cursor.CROSSHAIR_CURSOR, Cursor.TEXT_CURSOR y otros.

métodos de dibujo

método	uso
void paint(Graphics p**)**	Pinta el componente y sus subcomponentes. Delega sus funciones en los tres métodos siguientes

método	uso
void **paintComponent(**Graphics p**)**	Pinta sólo este componente. Este es el método recomendado en Swing para dibujar en un componente.
void **paintComponents(**Graphics p**)**	Llama a los métodos de pintura de todos los componentes de la ventana.
void **paintChildren(**Graphics p**)**	Pinta los componentes hijo de este componente
void **paintBorder(**Graphics p**)**	Pinta el borde del componente
protected Graphics **getComponentGraphics(**Graphics g**)**	Obtiene el objeto gráfico utilizado para dibujar el componente. El argumento es el objeto gráfico original. El otro es el tratado por el componente.
void **update(**Graphics g**)**	Llama a paint

activar y desactivar componentes

método	uso
void **setEnabled(**boolean activar**)**	Si el argumento es **true** se habilita el componente, si no, se deshabilita. Un componente deshabilitado es un método que actúa con el usuario. Por deshabilitar un componente, no se deshabilitan los hijos.

enfocar

Para que un componente sea al que van dirigidas las pulsaciones de las teclas o, dicho de otra forma, el que recibe la interacción del usuario, debe poseer el enfoque (**focus**).

En muchos casos, el enfoque salta de un control al siguiente pulsando la tecla tabulador. Varios métodos se encargan de controlar ese salto:

método	uso
void **requestFocus()**	Pide el foco para el componente. Será posible, si este es visible, activado y enfocable *focusable*). El contenedor del foco también poseerá el foco.
boolean **requestFocusInWindow()**	Pide el foco para el componente si la ventana contenedora poseía el foco. Devuelve **true** si el foco se puede dar sin problemas. Aunque sólo el evento FOCUS_GAINED es el encargado de indicar que el foco ha sido pasado. Actualmente, debido a que el método anterior es dependiente de la plataforma, se recomienda este método siempre que sea posible.
void **transferFocus()**	Hace que el siguiente componente en la lista de tabulaciones, obtenga el foco

método	uso
void **transferFocusBackward()**	El foco pasa al anterior componente en la lista de tabulaciones.
void **setNextFocusableComponent(** Component c**)**	Hace que el componente **c** sea el siguiente en la lista de tabulaciones.
Component **getNextFocusableComponent()**	Obtiene el siguiente componente de la lista de tabulaciones.

Contenedores

Son un tipo de componentes pensados para almacenar y manejar otros componentes. Los objetos JComponent pueden ser contenedores al ser una clase que desciende de **Container** que es la clase de los objetos contenedores de AWT.

Para hacer que un componente forme parte de un contenedor, se utiliza el método **add**. Mientras que el método **remove** es el encargado de eliminar un componente. Ambos métodos proceden de la clase **java.awt.Container**

Swing posee algunos contenedores especiales. Algunos son:

- **JWindow.** Representa un panel de ventana sin bordes ni elementos visibles.
- **JFrame.** Objeto que representa una ventana típica con bordes, botones de cerrar, etc.
- **JPanel.** Es la clase utilizada como contenedor genérico para agrupar componentes.
- **JDialog.** Clase que genera un cuadro de diálogo.
- **JApplet.** Contenedor que agrupa componentes que serán mostrados en un navegador.

componentes de un contenedor

Estos métodos de la clase **Container** permiten obtener información sobre componentes de un contenedor:

método	uso
Component[] **getComponents()**	Devuelve un array de componentes con todos los componentes correspondientes al contenedor actual
void **list(**PrintWriter out**)**	Escribe en el objeto **PrintWriter** indicado, la lista de componentes.
Component **getComponentAt(**int x, int y**)**	Indica qué componente se encuentra en esas coordenadas (calculadas dentro del sistema de coordenadas del contenedor).

JWindow

Este objeto deriva de la clase **java.awt.Window** que a su vez deriva de **java.awt.Container.** Se trata de un objeto que representa un marco de ventana

simple, sin borde, ni ningún elemento. Sin embargo son contenedores a los que se les puede añadir información. Estos componentes suelen estar dentro de una ventana de tipo Frame o, mejor, JFrame.

constructores

método	uso
JWindow()	Crea un marco de ventana típico e independiente
JWindow(Frame padre**)**	Crea un marco de ventana dentro de la ventana tipo *Frame* indicada.
JWindow(Window padre**)**	Crea un marco de ventana dentro de la ventana indicada.
JWindow(GraphicsConfiguration gc**)**	Crea una nueva ventana usando la configuración gráfica indicada
JWindow(Window padre, GraphicsConfiguration gc**)**	Crea una ventana dentro de la padre con la configuración de gráficos indicada

JFrame

Los objetos JFrame derivan de la clase Frame que, a su vez deriva, también de la clase Window, por lo que muchos métodos de esta clase son comunes a la anterior. Los objetos JFrame son ventanas completas.

constructores

constructor	descripción
JFrame()	Crea una ventana con la configuración normal.
JFrame(GraphicsConfiguration gc**)**	Crea una nueva ventana usando la configuración gráfica indicada
JFrame(String titulo**)**	Crea una ventana con el título indicado.
JFrame(String titulo, GraphicsConfiguration gc**)**	Crea una ventana con el título y la configuración gráfica indicada.

métodos

método	descripción
void **setDefaultCloseOperation(**int modo**)**	Indica el modo de cerrar la ventana, en sustitución del entero se puede utilizar los siguientes valores: ◉ JFrame**.EXIT_ON_CLOSE.** Al cerrar la ventana, se acaba el programa (desde versión 1.3 de Java) ◉ JFrame**.DO_NOTHING_ON_CLOSE.** No se hace nada al cerrar la ventana ◉ JFrame**.HIDE_ON_CLOSE.** Esconde la ventana al cerrarla. ◉ JFrame**.DISPOSE_ON_CLOSE.** Esconde la ventana y se deshace de ella.

método	descripción
	El programa acaba cuando todas las ventanas han sido cerradas con esta opción.
voit setResizable(boolean b**)**	Con **true** la ventana es cambiable de tamaño, en **false** la ventana tiene tamaño fijo.
void setTitle(String t**)**	Cambia el título de la ventana
void pack()	Ajusta la ventana a tamaño suficiente para ajustar sus contenidos
void setState(int estado)	Coloca la ventana en uno de estos dos estados: ◉ **Frame.NORMAL** ◉ **Frame.ICONOFIED.** Minimizada
void setExtendedState(int estado)	Idéntica a la anterior pero añade tres estados: ◉ **JFrame.MAXIMIZED_HOR.** Maximizada horizontal ◉ **JFrame.MAXIMIZED_VER.** Maximizada vertical ◉ **JFrame.MAXIMIZED_BOTH** Maximizada completa Sólo funciona desde la versión 1.4
int getState()	Indica el estado de la ventana
int getExtendedState()	Indica el estado de la ventana. Desde la versión 1.4

JDialog

JDialog deriva de la clase AWT Dialog que es subclase de Window. Representa un cuadro de diálogo que es una ventana especializada para realizar operaciones complejas.

constructores

método	uso
JDialog()	Crea una ventana con la configuración normal.
JDialog(Frame propietaria**)**	Crea un nuevo cuadro de diálogo, indicando como padre la ventana seleccionada
JDialog(Frame propietaria, **boolean** modal**)**	Crea un nuevo cuadro de diálogo, indicando como padre la ventana seleccionada. Poniendo a **true** el segundo parámetro, el cuadro de diálogo pasa a ser modal. Una ventana modal obliga a el usuario a contestar al cuadro antes de que pueda continuar trabajando.
JDialog(Frame propietaria, **String** título**)**	Crea un cuadro de diálogo perteneciente a la ventana indicada y poniendo el título deseado

método	uso
JDialog(Frame propietaria, String título, boolean modal**)**	Crea un cuadro de diálogo perteneciente a la ventana indicada, poniendo el título deseado e indicando si se desea el cuadro en forma modal.
JDialog(Frame propietaria, String título, boolean modal, GraphicsConfiguration gc**)**	Lo mismo, pero además indicando una posible configuración gráfica.

métodos interesantes

Hay varios métodos que se pueden usar con los objetos JDialog y JFrame

método	uso
void **toBack()**	Coloca la ventana al fondo, el resto de ventanas aparecen por delante
void **toFront()**	Envía la ventana al frente (además le dará el foco).
void **setTitle(**String t**)**	Cambia el título de la ventana
String **getTitle()**	Obtiene el título de la página
voit **setResizable(**boolean b**)**	Con **true** la ventana es cambiable de tamaño, en **false** la ventana tiene tamaño fijo.
void **pack()**	Ajusta la ventana a tamaño suficiente para ajustar sus contenidos

añadir componentes a las ventanas

Las clases JDialog y JFrame no permiten usar el método **add**, como les ocurre a los contenedores normales, por eso se utiliza el método **getContentPane()** que devuelve un objeto **Container** que representa el área visible de la ventana. A este contenedor se le llama panel contenedor y sí permite método **add**.

```
public class prbVentana{
  public static void main(String args[]){
        JFrame ventana=new JFrame("Prueba");
        ventana.setLocation(100,100);
        Container c=ventana.getContentPane();
        c.add(new JLabel("Hola"));
        ventana.pack();
        ventana.setVisible(true);
  }
```

Este código muestra una ventana ajustada al contenido de una ventana que pone Hola.

eventos

En términos de Java, un evento es un objeto que es lanzado por un objeto y enviado a otro objeto llamado *escuchador* (listener). Un evento se **lanza** (o se **dispara, *fire***) cuando ocurre una determinada situación (un clic de ratón, una pulsación de tecla,...).

La programación de eventos es una de las bases de Java y permite mecanismos de diseño de programas orientados a las acciones del usuario. Es decir, son las acciones del usuario las que desencadenan mensajes entre los objetos (el flujo del código del programa se desvía en función del evento producido, alterando la ejecución normal).

Hay multitud de tipos de eventos, más adelante se señala una lista de los eventos fundamentales. En su captura hay que tener en cuenta que hay tres objetos implicados:

- **El objeto fuente.** Que es el objeto que lanza los eventos. Dependiendo del tipo de objeto que sea, puede lanzar unos métodos u otros. Por ejemplo un objeto de tipo **JLabel** (etiqueta) puede lanzar eventos de ratón (**MouseEvent**) pero no de teclado (**KeyEvent**). El hecho de que dispare esos eventos no significa que el programa tenga que, necesariamente, realizar una acción. Sólo se ejecuta una acción si hay un objeto escuchando.

- **El objeto escuchador u oyente (*listener*).** Se trata del objeto que recibe el evento producido. Es el objeto que captura el evento y ejecuta el código correspondiente. Para ello debe implementar una interfaz relacionada con el tipo de evento que captura. Esa interfaz obligará a implementar uno o más métodos cuyo código es el que se ejecuta cuando se dispare el evento.

- **El objeto de evento.** Se trata del objeto que es enviado desde el objeto fuente a el escuchador. Según el tipo de evento que se haya producido se ejecutará uno u otro método en el escuchador.

escuchadores de eventos

Cada tipo de evento tiene asociado un interfaz para manejar el evento. A esos interfaces se les llama *escuchadores* (*Listeners*) ya que proporcionan métodos que están a la espera de que el evento se produzca. Cuando el evento es disparado por el objeto fuente al que se estaba escuchando, el método manejador del evento se dispara automáticamente.

Por ejemplo, el método **actionPerformed** es el encargado de gestionar eventos del tipo **ActionEvent** (eventos de acción, se producen, por ejemplo, al hacer clic en un botón). Este método está implementado en la interfaz **ActionListener** (implementa escuchadores de eventos de acción).

Cualquier clase que desee escuchar eventos (los suyos o los de otros objetos) debe implementar la interfaz (o interfaces) pensada para capturar los eventos del tipo deseado. Esta interfaz habilita a la clase para poder implementar métodos de gestión de eventos.

Por ejemplo; un objeto que quiera escuchar eventos **ActionEvent**, debe implementar la interfaz **ActionListener.** Esa interfaz obliga a definir el método ya comentado **actionPerformed**. El código de ese método será invocado automáticamente cuando el objeto fuente produzca un evento de acción.

Es decir, hay tres actores fundamentales en el escuchador de eventos:

- El **objeto de evento** que se dispara cuando ocurre un suceso. Por ejemplo para capturar el ratón sería **MouseEvent**.
- El **método o métodos de captura del evento** (que se lanza cuando el evento se produce). Pueden ser varios, por ejemplo para la captura de eventos de tipo **MouseEvent** (evento de ratón) existen los métodos **mouseReleased** (es invocado cuando se libera un botón del ratón), **mousePressed** (es invocado cuando se pulsa un botón del ratón), **mouseEntered** (es invocado cuando el cursor entra en el objeto) y **mouseExited** (ocurre cuando el ratón sale del objeto).
- La **interfaz** que tiene que estar implementada en la clase que desea capturar ese evento. En este ejemplo sería **MouseListener,** que es la que obliga a la clase del escuchador a implementar los cuatro métodos de gestión comentados anteriormente

Sin duda, el más complejo es este último, pero hay que entender que una interfaz lo único que consigue es dar a una clase la facultad de escuchar (***Listen***) eventos.

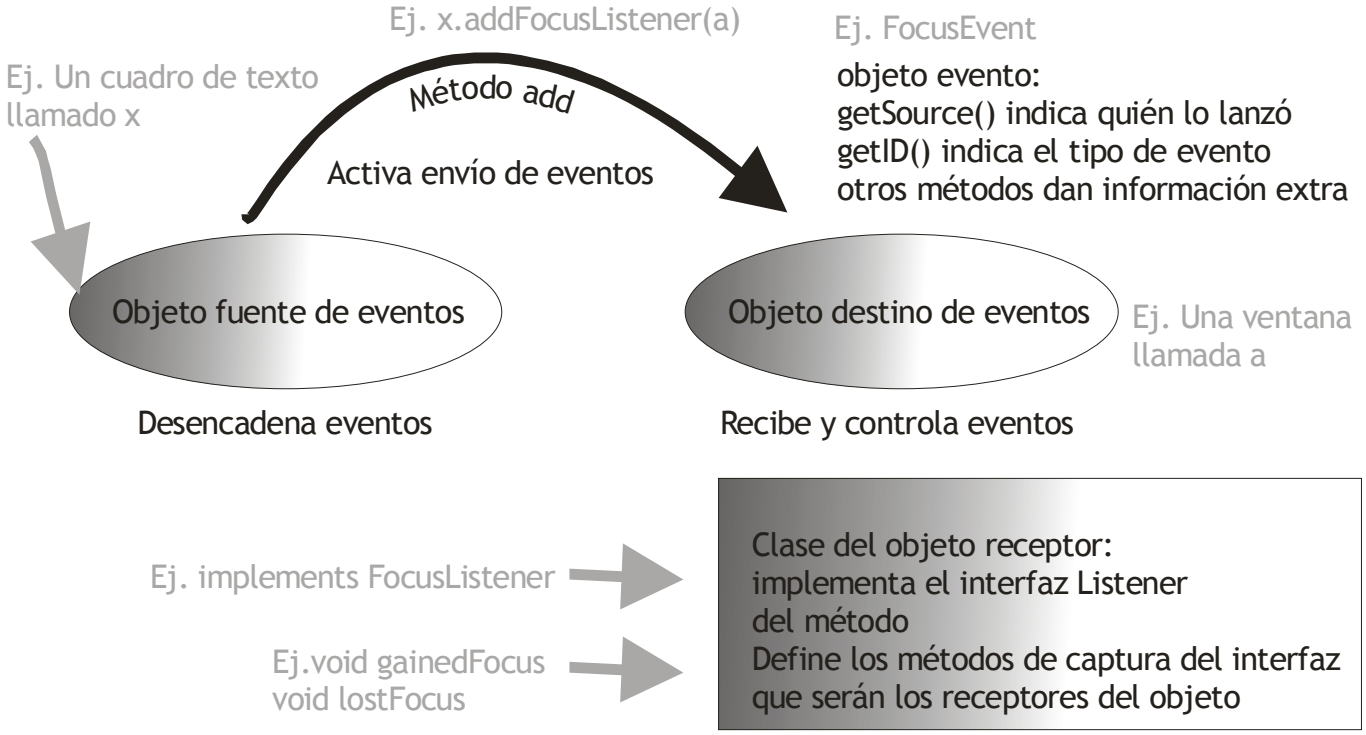


Ilustración 20, Proceso de gestión de eventos

fuentes de eventos

disparar eventos

El objeto fuente permite que un objeto tenga capacidad de enviar eventos. Esto se consigue mediante un método que comienza por la palabra **add** seguida por el nombre

de la interfaz que captura este tipo de eventos. Este método **recibe como parámetro el objeto escuchador de los eventos**.

Esto es más fácil de lo que parece. Para que un objeto fuente, sea escuchado, hay que indicar quién será el objeto que escuche (que obligadamente deberá implementar la interfaz relacionada con el evento a escuchar). Cualquier componente puede lanzar eventos, sólo hay que indicárselo, y eso es lo que hace el método add. Ejemplo:

```
public class MiVentana extends JFrame implements ActionListener{
   JButton boton1=new JButton("Prueba");
   //Constructor
   public MiVentana() {
         ...
         boton1.addActionListener(this);//El botón lanza
               //eventos que son capturados por la ventana
         ...
   }
   ...
   public void actionPerformed(ActionEvent e){
         //Manejo del evento
   }
}
```

En el ejemplo anterior se habilita al *boton1* para que lance eventos mediante el método **addActionListener**. Este método requiere un objeto escuchador que, en este caso, será la ventana en la que está el botón. Esta ventana tiene que implementar la interfaz **ActionListener** para poder escuchar eventos (de hecho el método *addActionListener* sólo permite objetos de esta interfaz). Cuando se haga clic con el ratón se llamará al método **actionPerformed** de la ventana, que es el método de gestión.

Hay que señalar que una misma fuente puede tener varios objetos escuchando los eventos (si lanza varios métodos *add*). Si hay demasiados objetos escuchando eventos, se produce una excepción del tipo **TooManyListenersException**

eliminar oyentes

Hay un método **remove** que sirve para que un oyente del objeto deje de escuchar los eventos.

```
boton1.removeActionListener(this); //La ventana deja de
                  //escuchar los eventos del botón
```

objeto de evento.

clase *EventObject*

Ya se ha comentado que cuando se produce un evento se crea un objeto llamado **objeto de evento.** Este objeto es pasado al objeto que está *escuchando* los eventos.

Todos los objetos de evento pertenecen a clases que derivan de **EventObject .** Esta es la superclase de todos los objetos de evento. Representa un evento genérico y en la práctica sólo sirve para definir los métodos comunes a todos los eventos que son:

método	uso
Object **getSource()**	Obtiene el objeto que lanzó el evento (método muy importante)
String **toString()**	Método toString redefinido para mostrar la información del evento

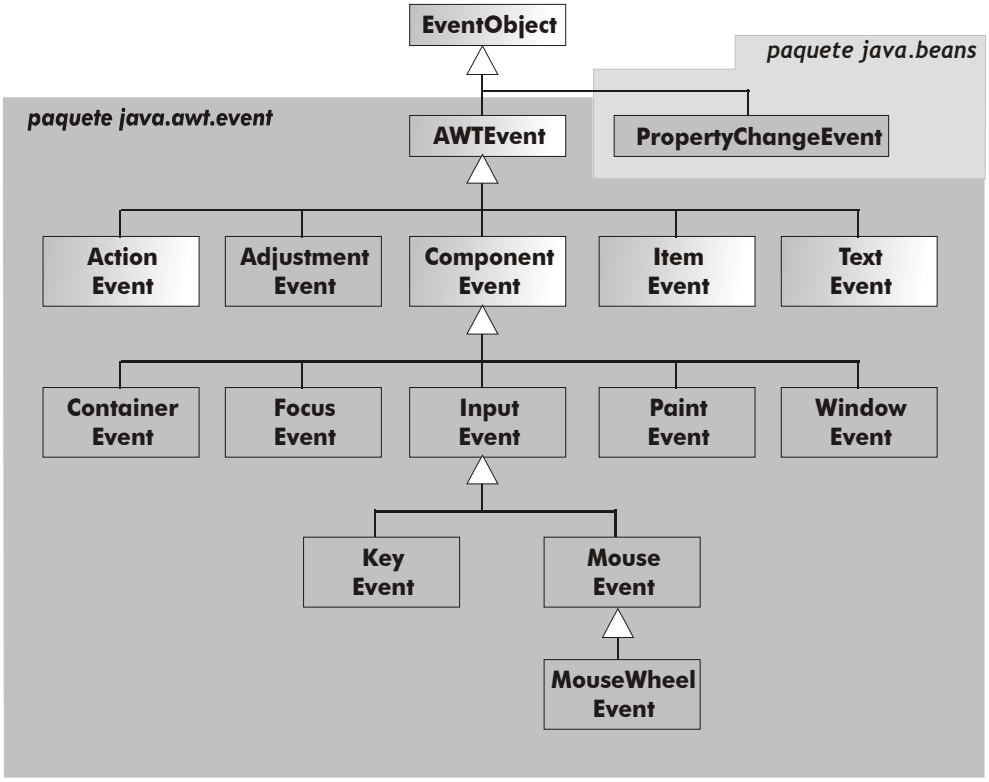


Ilustración 21, Jerarquía de los objetos de evento

clase *AWTEvent*

Se trata de una clase descendiente de **EventObject** y padre de todos los eventos para componentes Swing y AWT. Se encuentra en el paquete **java.awt.event.** Proporciona métodos comunes para todas sus clases hijas. Estos son (además de los ya comentados **getSource** y **toString**):

método	uso
void **getID()**	Obtiene el identificador de tipo de evento
String **paramString()**	Cadena de estado del evento (se usa sólo para depuración)
void **setSource(**Object o**)**	Redirige el evento al objeto *o*, haciendo que sea éste el que lance de nuevo el evento.

método	uso
protected void **consume()**	Consume el evento. Esto es, hace que el evento deje de ser enviado. No todos los eventos lo permiten
protected boolean **isConsumed()**	Indica si el evento ya ha sido consumido

eventos InputEvent

Es una de las clases de eventos fundamentales, deriva de **ComponentEvent**. Hay varios tipos de evento que derivan de éste. Se trata de los eventos **KeyEvent** y **MouseEvent**. La clase **InputEvent** viene con unas serie de indicadores que permiten determinar qué teclas y/o botones de ratón estaban pulsados en el momento del evento. El método **getModifiers** devuelve un entero que permite enmascarar con esas constantes para determinar las teclas y botones pulsados.

Este enmascaramiento se realiza con el operador lógico AND (**&**) en esta forma:

```
public void mouseReleased(MouseEvent e){
   int valor=e.getModifiers();
   if((valor & InputEvent.SHIFT_MASK)!=0){
         //La tecla Mayúsculas (Shift) se pulsó con el
         //ratón
```

Constantes de máscaras predefinidas en la clase **InputEvent**: SHIFT_MASK, ALT_MASK, ALT_GRAPH_MASK, BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK. Actualmente se utiliza otro formato que incluye la palabra DOWN. Es decir: ALT_DOWN_MASK, ALTA_GRAPH_DOWN_MASK, etc. Estas últimas aparecieron en la versión 1.4. y se utilizan con el método **getModifiersExt.**

También se pueden utilizar los métodos **isShiftDown** (mayúsculas pulsada), **isAltDown** (Alt pulsada), **isAltGraphDown** (Alt Graph pulsada) o **isControlDown** (tecla control). Todos devuelven **true** si la tecla en cuestión está pulsada. La lista completa de métodos de esta clase es:

método	uso
void **consume()**	Hace que el evento de entrada deje de ser enviado. Es decir, le consume (elimina el evento). Deriva de **AWTEvent**, pero en este caso es un método público.
boolean **isConsumed()**	Indica si el evento ya ha sido consumido
int **getModifiers()**	Descrita anteriormente. Devuelve un entero que representa una máscara de bits donde los bits representan con valor 1, teclas de control y botones de ratón pulsado. Por lo que hay que comparar con máscaras.
int **getModifiersExt()**	Versión moderna de la anterior (comentada anteriormente). Apareció en la versión 1.4

método	uso
long **getWhen()**	Devuelve el momento exacto en el que ocurrió el evento. Lo devuelve en formato **long** (que representa los milisegundos acaecidos). Habría que convertirlo a objeto **Date** (mediante el constructor ***new Date(long)***) para así poder obtener hora, minutos y segundos (y también la fecha)
boolean **isAltDown()**	Devuelve **true** si la tecla Alt estaba pulsada cuando se produjo el evento.
boolean **isAltGraphDown()**	Devuelve **true** si la tecla Alt estaba pulsada cuando se produjo el evento.
boolean **isAltDown()**	Devuelve **true** si la tecla Alterno Gráfico (*Alt Gr*) estaba pulsada cuando se produjo el evento.
boolean **isControlDown()**	Devuelve **true** si la tecla Control estaba pulsada cuando se produjo el evento.
boolean **isShiftDown()**	Devuelve **true** si la tecla Mayúsculas (*Shift*) estaba pulsada cuando se produjo el evento.

lista completa de eventos

Objeto de evento	java.awt.event.**ComponentEvent** Ocurre cuando un componente se mueve, cambia de tamaño o de visibilidad
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	void **addComponentListener(**ComponentListener objeto**)**
Interfaz de gestión	java.awt.event.**ComponentListener**
Métodos de control del evento (receptores del objeto de evento)	void **componentResized(**ComponentEvent e**)** Se produce cuando el componente cambió de tamaño void **componentMoved(**ComponentEvent e**)** Se produce cuando el componente se movió void **componentShown(**ComponentEvent e**)** Se produce cuando el componente se mostró void **componentHidden(**ComponentEvent e**)** Se produce cuando el componente cambió se puesooculto
Métodos del objeto *ComponentEvent* interesantes	Component **getComponent()** Obtiene el componente que generó el evento

Objeto de evento	**java.awt.event.FocusEvent** Ocurre cuando cambia el foco de un componente
Ocurre en	**Cualquier componente**
Método lanzador	**void addFocusListener(FocusListener** objeto**)**
Interfaz de gestión	**java.awt.event .FocusListener**
Métodos de control	**void focusGained(FocusEvent** e**)** Ocurre cuando se obtiene el foco **void focusLost(FocusEvent** e**)** Ocurre cuando se pierde el foco
Métodos del objeto *FocusEvent* interesantes	**Component getOppositeComponent()** Obtiene el otro componente involucrado en el foco. Si el objeto actual gana el foco, el objeto involucrado es el que lo perdió. **boolean isTemporary()** Con valor **true** indica que el cambio de foco es temporal.

Objeto de evento	**java.awt.event.KeyEvent** Ocurre cuando se pulsa una tecla (y el componente posee el foco). Es un evento del tipo **InputEvent**
Ocurre en	Cualquier **componente**
Método lanzador	**void addKeyListener(KeyListener** objeto**)**
Interfaz de gestión	**java.awt.event.KeyListener**
Métodos de control	**void keyTyped(KeyEvent** e**)** Ocurre cuando se pulsa una tecla que escribe (una tecla que escriba un símbolo Unicode, la tecla F1 por ejemplo no valdría) **void keyPressed(KeyEvent** e**)** Ocurre cuando se pulsa cualquier tecla **void keyReleased(KeyEvent** e**)** Ocurre cuando se levanta cualquier tecla
Métodos del objeto *KeyEvent* interesantes	**char getKeyChar**() Devuelve el código Unicode de la tecla pulsada **int getKeyCode**() Devuelve el código entero de la tecla pulsada, útil para capturar teclas de control. Se pueden utilizar constantes del objeto *KeyEvent* para comparar. Esas constantes empiezan con el signo VK_ seguida del nombre de la tecla en mayúsculas, por ejemplo KeyEvent.VK_F1 **representa** la tecla F1. **char getKeyText(int** código) Devuelve la traducción del código a tecla a formato de texto **boolean isActionKey()** Vale true si el evento lo provocó una tecla de acción. Una tecla de acción es una tecla que no escribe y que no modifica otras teclas (como *Shift* o *Alt*)

Objeto de evento	**java.awt.event.MouseEvent** Ocurre cuando se pulsa el ratón sobre un componente. Es un evento del tipo **InputEvent**
Objetos que le pueden disparar	Cualquier componente
Método lanzador de eventos	**void addMouseListener(MouseListener** objeto**)**
Interfaz de gestión	**java.awt.event.MouseListener**
Métodos de control	**void mouseClicked (MouseEvent e)** Se hizo clic con un botón del ratón en el componente **void mousePressed(MouseEvent e)** Se pulsó un botón del ratón **void mouseReleased(MouseEvent e)** Se levantó un botón del ratón **void mouseEntered(MouseEvent e)** El ratón está sobre el componente. **void mouseExited(MouseEvent e)** El ratón salió fuera de el componente
Métodos del objeto *MouseEvent* interesantes	**int getX()** Devuelve la posición X de pantalla del ratón **int getY()** Devuelve la posición Y de pantalla del ratón **Point getPoint()** Devuelve la posición del cursor de ratón en forma de objeto Point. **int getClickCount()** Devuelve el número de clics seguidos realizados por el ratón **int getButton()** Indica qué botón del ratón se pulsó. Se pueden usar estas constantes: **MouseEvent.BUTTON1, MouseEvent.BUTTON2** y **MouseEvent.BUTTON3, MouseEvent.NOBUTTON.** Funciona desde la versión 1.4 de Java.

Tipo de evento	***MouseMotion*** Sirve para capturar objetos **MouseEvent** igual que en el caso anterior, solo que en este caso permite capturar eventos de arrastre y movimiento del ratón.
Ocurre en	Cualquier componente
Método lanzador	void **addMouseMotionListener(**MouseMotionListener objeto**)**
Interfaz de gestión	java.awt.event **MouseMotionListener**
Métodos de control	void **mouseDragged(**MouseEvent e**)** Ocurre cuando se arrastra con el ratón al componente void **mouseMoved(**MouseEvent e**)** Ocurre siempre que el ratón se mueva en el componente

Objeto de evento	java.awt.event.**MouseWheelEvent** Sirve para capturar eventos de movimiento de rueda de ratón (en los ratones que posean esta característica). Implementado en la versión 1.4
Ocurre en	Cualquier componente
Método lanzador	void **addMouseWheelListener(**MouseWheelListener objeto**)**
Interfaz de gestión	java.awt.event **MouseMotionListener**
Métodos de control	void **mouseWheelMoved(**MouseEvent e**)** Ocurre cuando se ha movido la rueda del ratón
Métodos del objeto *MouseEvent* interesantes	int **getWheelRotation()** indica lo que se ha movido la rueda del ratón. Devuelve valores positivos o negativos según la dirección del movimiento de la rueda. int **getScrollType()** Devuelve un valor que indica si el ratón realiza o no movimientos de desplazamiento. Los valores pueden ser: ◉ **MouseWheelEvent.WHEEL_UNIT_SCROLL.** Significa que funciona el scroll ◉ **MouseWheelEvent.WHEEL_BLOCK_SCROLL.** El giro de la rueda está bloqueado. int **getScrollAmount()** Devuelve el valor de desplazamiento de la rueda cada vez que gira. Sólo es válido su resultado si el método anterior devuelve MouseWheelEvent.WHEEL_UNIT_SCROLL Además este objeto deriva del anterior (MouseEvent) por lo que posee sus mismos métodos.

Objeto de evento	java.awt.event.**WindowEvent** Captura eventos de ventana
Objetos que le pueden disparar	**JDialog, JWindow, JFrame**
Método lanzador	void **addWindowListener(**WindowListener objeto**)**
Interfaz de gestión	java.awt.event.**WindowListener**
Métodos de control	void **windowOpened(**WindowEvent e**)** La ventana se está abriendo void **windowClosing(**WindowEvent e**)** La ventana se está cerrando void **windowClosed(**WindowEvent e**)** La ventana se cerró void **windowIconified(**WindowEvent e**)** Se minimizó la ventana void **windowDeiconified(**WindowEvent e**)** Se restauró la ventana void **windowActivated(**WindowEvent e**)** Se activó la ventana

Objeto de evento	java.awt.event.**ContainerEvent** Ocurre cuando se añaden o quitan controles a un contenedor del tipo que sea
Objetos que le pueden disparar	**Cualquier contenedor**
Método lanzador de eventos	void **addContainerListener(**ContainerListener objeto**)**
Interfaz de gestión	**ContainerListener**
Métodos de control	void **componentAdded(**ContainerEvent e**)** Cuando se añade un componente void **componentRemoved(**ContainerEvent e**)** Cuando se quita un componente

Objeto de evento	**java.awt.event.ActionEvent** Se ejecuta una orden de acción
Objetos que le pueden disparar	**JButton, JCheckBoxMenuItem, JComboBox, JFileChooser, JRadioButtonItem, JTextField, JToggleButton**
Método lanzador de eventos	**void addActionListener(**ActionListener capturador**)**
Interfaz de gestión	**ActionListener**
Métodos de control	**void performedAction(**ActionEvent e**)** Cuando se efectúa una acción
Métodos del objeto *ActionEvent* interesantes	**String getActionCommand()** Devuelve una cadena que identifica la acción

Objeto de evento	**java.awt.event.AdjustmentEvent** Ocurre cuando se mueve el scroll de la pantalla
Objetos que le pueden disparar	**JScrollBar**
Método lanzador de eventos	**void addAdjustmentListener(**AdjustmentListener capturador)
Interfaz de gestión	**AdjustmentListener**
Métodos de control	**void adjustmentValueChange(**AdjustmentEvent e**)** Cuando cambia el valor de una barra de desplazamiento
Métodos interesantes del objeto *AdjustmentEvent*	**int getAdjustmentType()** Devuelve un entero que indica el tipo de cambio en el desplazamiento de ventana. Se puede comparar con las constantes estáticas de esta clase llamadas: UNIT_INCREMENT, UNIT_DECREMENT BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK **int getValue()** Devuelve el valor de cambio en las barras **Adjustable getAdjustable()** Obtiene el objeto de tipo *Adjustable* que originó el evento.

Objeto de evento	javax.swing.event.**HyperlinkEvent** Ocurre cuando le sucede algo a un texto de enlace (**hipervínculo**), un clic, posar el ratón,...
Objetos que le pueden disparar	**JEditorPane, JTextPane**
Método lanzador de eventos	void **addHyperlinkListener(**HyperlinkListener capturador**)**
Interfaz de gestión	**HyperlinkListener**
Métodos de control	void **hyperlinkUpdate(**HyperlinkEvent e**)** Ocurre cuando se actúa sobre un hipervínculo
Métodos interesantes del objeto evento	URL **getURL()** Devuelve la dirección URL del enlace String **getDescripción()** Devuelve el texto del enlace HyperlinkEvent.EventType **getEventType()** Devuelve un objeto que permite saber que tipo de evento se realizó sobre el enlace

Objeto de evento	java.awt.event.**ItemEvent** Ocurre cuando se cambia el estado de un control de tipo ítem
Objetos que le pueden disparar	**JCheckBoxmenuItem, JComboBox, JRadioButtonMenuItem**
Método lanzador de eventos	void **addItemListener(**ItemListener capturador**)**
Interfaz de gestión	**ItemListener**
Métodos de control	void **itemStateChanged(**ItemEvent e**)** Cuando se cambió un item
Métodos interesantes del objeto evento	Object **getItem()** Obtiene el elemento que originó el evento int **StateChange()** Devuelve el tipo de cambio que se produjo con el ítem. Se puede comparar con las constantes **ItemEvent.SELECTED** y **ItemEvent.DESELECTED**

Objeto de evento	javax.swing.event.**ListSelectionEvent** Cambio en la selección de un control de lista
Objetos que le pueden disparar	**JList**
Método lanzador de eventos	void **addListSelectionListener(**ListSelectionListener cap**)**
Interfaz de gestión	**ListSelectionListener**
Métodos de control	void **valueChanged(**ListSelectionEvent e**)** Cuando se cambió valor de la lista
Métodos interesantes del objeto *ListSelectionEvent*	int **getFirstIndex()** Número de la primera fila cambiada con el evento int **getLastIndex()** Número de la última fila cambiada con el evento

Objeto de evento	javax.swing.event.**MenuEvent** Cambio en la selección de un menú
Objetos que le pueden disparar	**JMenu**
Método lanzador	void **addMenuListener(**MenuListener capturador**)**
Interfaz de gestión	**MenuListener**
Métodos de control	void **menuCanceled(**MenuEvent e**)** Si se canceló el menú void **menuDeselected(**MenuEvent e**)** Si se deseleccionó el menú void **menuSelected(**MenuEvent e**)** Si se seleccionó el menú

Objeto de evento	javax.swing.event.**PopupMenuEvent** Cambio en la selección de un menú de tipo *popup*
Objetos que le pueden disparar	**JPopupMenu**
Método lanzador	void **addPopupMenuListener(**PopupMenuListener e**)**
Interfaz de gestión	**PopupMenuListener**
Métodos de control	void **popupMenuCanceled(**PopupMenuEvent e**)** Si se canceló el menú void **popupMenuWillBecomeInvisible(**PopupMenuEvent e**)** Si va a desaparecer el menú void **popupMenuWillBecomeVisible(**PopupMenuEvent e**)** Si va a aparecer el menú

Objeto de evento	**javax.swing.event.MenuKeyEvent** Ocurre cuando se pulsaron teclas hacia el menú
Objetos que le pueden disparar	**JMenuItem**
Método lanzador	**void addMenuKeyListener(MenuKeyListener** capturador**)**
Interfaz de gestión	**MenuKeyListener**
Métodos de control	**void popupMenuKeyPressed(MenuKeyEvent** e**)** Se pulsó la tecla **void popupMenuKeyReleased(MenuKeyEvent** e**)** Se levantó la tecla **void popupMenuKeyTyped(MenuKeyEvent** e**)** Se pulsó y levantó una tecla de escritura

Objeto de evento	**javax.swing.event.MenuDragMouseEvent** Ocurre cuando se arrastró el ratón sobre un elemento de menú
Objetos que le pueden disparar	**JMenuItem**
Método lanzador	**void addMenuDragMouseListener(MenuDragMouseListener** c**)**
Interfaz de gestión	**MenuDragMouseListener**
Métodos de control	**void popupMenuDragMouseDragged(MenuDragMouseEvent** e**)** El ratón está arrastrando sobre el menú **void popupMenuDragMouseEntered(MenuDragMouseEvent** e**)** El ratón está arrastrando dentro del menú **void popupMenuDragMouseExited(MenuDragMouseEvent** e**)** El ratón está arrastrando hacia fuera del menú **void popupMenuDragMouseReleased(MenuDragMouseEvent** e**)** Se soltó el ratón que se arrastraba dentro del menú

Objeto de evento	javax.swing.event.**ChangeEvent** Cambio en la selección de un control de lista
Objetos que le pueden disparar	**JSlider, JTabbedPane, JSpinner**
Método lanzador de eventos	void **addChangeListener(**ChangeListener cap**)**
Interfaz de gestión	**ChangeListener**
Métodos de control	void **stateChanged(**ChangeEvent e**)** Se produce cuando se cambió el valor del elemento

Objeto de evento	java.awt.event. **InternalFrameEvent** Ocurre cuando hay cambios en objetos de ventana interna
Objetos que le pueden disparar	**JInternalFrame**
Método lanzador de eventos	void **addInternalFrameListener(**InternalFrameListener cap**)**
Interfaz de gestión	**InternalFrameListener**
Métodos de control	void **internalFrameActivated(**InternalFrameEvent e**)** Se activa la ventana void **internalFrameClosed(**InternalFrameEvent e**)** Se cierra la ventana void **internalFrameClosed(**InternalFrameEvent e**)** Se cierra la ventana void **internalFrameDeactivated(**InternalFrameEvent e**)** Se desactiva la ventana void **internalFrameDeiconified(**InternalFrameEvent e**)** Se restaura la ventana void **internalFrameIconified(**InternalFrameEvent e**)** Se minimiza la ventana void **internalFrameOpened(**InternalFrameEvent e**)** Se abre la ventana
Métodos interesantes del objeto *InternalFrameEvent*	InternalFrame **getInternalFrame()** Devuelve la ventana interna que originó el evento

lanzar eventos propios

Se pueden crear eventos propios y lanzarlos a cualquier objeto que esté preparado para capturar eventos. Para ello basta crear el evento deseado indicando, al menos, en el constructor el objeto que capturará el evento y el identificador de evento.

El identificador es un entero que sirve para indicar el tipo de evento producido. En un evento MouseEvent habrá que indicar si es un evento de clic (MouseEvent.MOUSE_CLICKED), de arrastre (MouseEvent.MOUSEDRAGGED,...). Además según el tipo de evento se pueden requerir más valores (posición del cursor, etc.).

En general está técnica sirve para hacer pruebas, pero también se emplea para otros detalles. Ejemplo:

```
ventana v1=new ventana();
v1.setLocation(100,100);
v1.setSize(300,300);
v1.setVisible(true);

WindowEvent we=new WindowEvent(v1,WindowEvent.WINDOW_CLOSING);
v1.dispatchEvent(we);
```

Suponiendo que *ventana* sea una clase preparada para escuchar eventos de tipo *WindowsEvent*, se crea el objeto de evento *we*. El envío del evento se realiza con el método **dispachEvent.**

adaptadores

Para facilitar la gestión de eventos en ciertos casos, Java dispone de las llamadas clases adaptadores. Gracias a ellas, en muchos casos se evita tener que crear clases sólo para escuchar eventos. Estas clases son clases de contenido vacío pero que son muy interesantes para capturas sencillas de eventos.

Todas poseen la palabra **adapter** en el nombre de clase. Por ejemplo esta es la definición de la clase **MouseAdapter:**

```
public abstract class MouseAdapter implements MouseListener
{
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

Es una clase que implementa el interfaz **MouseListener**, pero que no define lo que hace cada método de captura. Eso se suele indicar de manera dinámica:

```
JFrame ventana =new JFrame("prueba");
ventana.setLocation(100,100);
ventana.setSize(300,300);
ventana.setVisible(true);
ventana.addMouseListener(new MouseAdapter(){
            public void mouseClicked(MouseEvent e){
```

```
                    System.out.println("Hola");
                }};
```

En el ejemplo anterior al hacer clic en la ventana se escribe el mensaje *Hola* en la pantalla. No ha hecho falta crear una clase para escuchar los eventos. Se la crea de forma dinámica y se la define en ese mismo momento. La única función del adaptador es capturar los eventos deseados.

Otro ejemplo (hola mundo en Swing):

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

public class HolaMundoSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HolaMundoSwing");
        JLabel label = new JLabel("Hola Mundo");
        frame.getContentPane().add(label);
        frame.addWindowListener(new WindowAdapter(){
           public void windowClosing(WindowEvent e){
             System.exit(0);
           }
        }
        );

        frame.pack();
        frame.setVisible(true);
    }
}
```

El resultado de ese famoso código es esta ventana:

El evento *windowClosing* está capturado por una clase adaptadora, cuya efecto es finalizar el programa cuando se cierra la ventana.

Clases adaptadoras:

- **ComponentAdapter**
- **ContainerAdapter**
- **FocusAdapter**
- **InternalFrameAdapter**

- **KeyAdapter**
- **MouseAdapter**
- **MouseMotionAdapter**
- **PrintJobAdapter**
- **WindowAdapter**

mensajes hacia el usuario. clase JOptionPane

Una de las labores típicas en la creación de aplicaciones gráficas del tipo que sea, es la de comunicarse con el usuario a través de mensajes en forma de cuadro de diálogo. Algunos cuadros son extremadamente utilizados por su sencillez (textos de aviso, error, confirmación, entrada sencilla de datos, etc.).

La clase **JOptionPane** deriva de **JComponent** y es la encargada de crear este tipo de cuadros. Aunque posee constructores, normalmente se utilizan mucho más una serie de métodos estáticos que permiten crear de forma más sencilla objetos *JOptionPane*.

cuadros de información

Son cuadros de diálogo que sirven para informar al usuario de un determinado hecho. Se construyen utilizando los siguientes métodos estáticos:

métodos	uso
static void showMessageDialog(Component padre, Object mensaje**)**	Muestra un cuadro de diálogo en el contenedor *padre* indicado con un determinado mensaje
static void showMessageDialog(Component padre, Object mensaje, String título, int tipo**)**	Muestra un cuadro de diálogo en el contenedor *padre* indicado con un determinado mensaje, título y tipo.
static void showMessageDialog(Component padre, Object mensaje, String título, int tipo, Icon i**)**	Igual que el anterior pero se permite indicar un icono para acompañar el mensaje

Estos son los posibles creadores de este tipo de cuadro. El tipo puede ser una de estas constantes:

- JOptionPane.**INFORMATION_MESSAGE**.
- JOptionPane.**ERROR_MESSAGE**.
- JOptionPane.**WARNING_MESSAGE**. Aviso
- JOptionPane.**QUESTION_MESSAGE**. Pregunta
- JOptionPane.**PLAIN_MESSAGE**. Sin icono

Ejemplo:

```
JOptionPane.showMessageDialog(this, "Soy un mensaje normal",
"Cuadro 1", JOptionPane.INFORMATION_MESSAGE);
```

El resultado es:

cuadros de confirmación

La diferencia con los anteriores reside en que en estos hay que capturar la respuesta del usuario para comprobar si acepta o declina el mensaje. Los métodos estáticos de creación son:

métodos	uso
static int showConfirmDialog(Component padre, Object mensaje**)**	Muestra un cuadro de confirmación en el componente *padre* con el mensaje indicado y botones de **Sí, No** y **Cancelar**
static int showConfirmDialog(Component padre, Object mensaje, **String** título, int opciones**)**	Muestra cuadro de confirmación con el título y mensaje reseñados y las opciones indicadas (las opciones se describen al final)
static int showConfirmDialog(Component padre, Object mensaje, String título, int opciones, int tipo**)**	Como el anterior pero indicando el tipo de cuadro (los posibles valores son los indicados en la página anterior) y un icono
static int showConfirmDialog(Component padre, Object mensaje, String título, int opciones, int tipo, Icon icono**)**	Como el anterior pero indicando un icono.

Obsérvese como el tipo de retorno es un número entero; este número representa el botón del cuadro sobre el que el usuario hizo clic. Este valor se puede representar por medio de estas constantes de JOptionPane

- JOptionPane.**NO_OPTION**. El usuario no pulsó ningún botón en el cuadro
- JOptionPane.**CLOSE_OPTION**. El usuario cerró sin elegir nada
- JOptionPane.**OK_OPTION**. El usuario pulsó OK
- JOptionPane.**YES_OPTION**. El usuario pulsó el botón **Sí**
- JOptionPane.**CANCEL_OPTION**. El usuario pulsó el botón **Cancelar**

Estas otras constantes facilitan el uso del parámetro *opciones* que sirve para modificar la funcionalidad del cuadro. Son:

- JOptionPane.**OK_CANCEL_OPTION**. Cuadro con los botones OK y Cancelar
- JOptionPane.**YES_NO_OPTION**. Cuadro con botones Sí y No

- **JOptionPane.YES_NO_CANCEL_OPTION**. Cuadro con botones Sí, No y Cancelar

Ejemplo:

```
int res= JOptionPane.showConfirmDialog(null,
   "¿Desea salir?", "Salir",
   JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE);

if (res==JOptionPane.YES_OPTION) System.exit(0);
```

cuadros de diálogo para rellenar entradas

Son cuadros que permiten que el usuario, desde un mensaje de sistema, rellene una determinada variable.

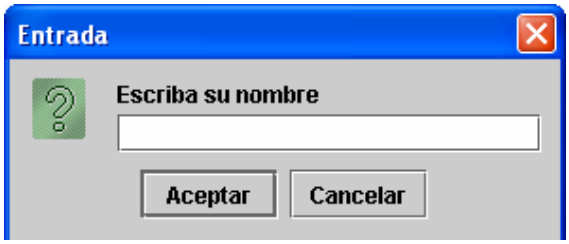
métodos	uso
static String **showInputDialog(**Object mensaje**)**	Muestra un cuadro de entrada con el mensaje indicado
static String **showInputDialog(** Component padre, Object mensaje**)**	Muestra un cuadro de entrada en el componente *padre* con el mensaje indicado
static **String showInputDialog(** Component padre, Object mensaje, String título, int tipo**)**	Muestra cuadro de entrada con el título y mensaje reseñados y el tipo que se indica
static Object **showInputDialog(** Component padre, Object mensaje, String título, int tipo, Icono icono, Object[] selección, Object selecciónInicial**)**	Indica además un icono, selecciones posibles y la selección inicial. El valor devuelto es un objeto **Object**.

Todos los métodos devuelven un String en el que se almacena la respuesta del usuario. En caso de que el usuario cancele el cuadro, devuelve **null** en la cadena a examinar.

Ejemplo:

```
String  res= JOptionPane.showInputDialog("Escriba su nombre");
if (res==null) JOptionPane.showMessageDialog(this,"No
escribió"); else nombre=res;
```

El resultado es:



Otra posibilidad es crear cuadros de entrada que utilicen controles de cuadro combinado en lugar de cuadros de texto. Para ello hay que preparar un array de Strings (u objetos convertibles a String) y pasarle al método de creación. Ejemplo:

```
String opciones[]={"España","`Italia", "Francia", "Portugal"};
String  res= (String)JOptionPane.showInputDialog(this,
  "Hola","Método",JOptionPane.INFORMATION_MESSAGE,null,
  opciones,opciones[0]);
```

cuadros de diálogo internos

Son cuadros que están dentro de un contenedor y no pueden salir fuera de él. Son más ligeros (ocupan menos recursos). Se crean con los mismos métodos y posibilidades que los anteriores, sólo que incorporan la palabra **Internal**. Funciones de creación:

- **showInternalMessageDialog.** Crea un mensaje normal, pero interno. Los parámetros son los mismos que **showMessageDialog.**
- **showInternalInputDialog.** Crea un mensaje interno de entrada de datos. Los parámetros son los mismos que **showInputDialog.**
- **showInternalConfirmDialog.** Crea un mensaje interno de confirmación. Los parámetros son los mismos que **showConfirmDialog.**

Programación de gráficos. Java2D

Java2D

En el tema anterior se habló del conjunto de librerías **JFC** (*Java Foundation Classes*), de la que formaba parte **Swing**. JFC es enorme y forma parte de ella también una API para la programación de gráficos en dos dimensiones (2D).

Anteriormente a Swing, los gráficos se dibujaban utilizando las clases AWT. En este tema se habla de clases Swing, pero se comentarán diferencias respecto a AWT.

paneles de dibujo

A la hora de hacer que los gráficos aparezcan en la pantalla, éstos se han de dibujar sobre un lienzo. El lienzo es una forma abstracta de indicar que hace falta un marco de trabajo. Lo malo es que normalmente una aplicación Java se sostiene sobre objetos que se sostienen en marcos como **JFrame** por ejemplo. Estos marcos principales en Swing implementan una interfaz llamada **RootPaneContainer** que coloca varios paneles superpuestos a fin de organizar la información. Las clases que implementan esta estructura son las siguientes:

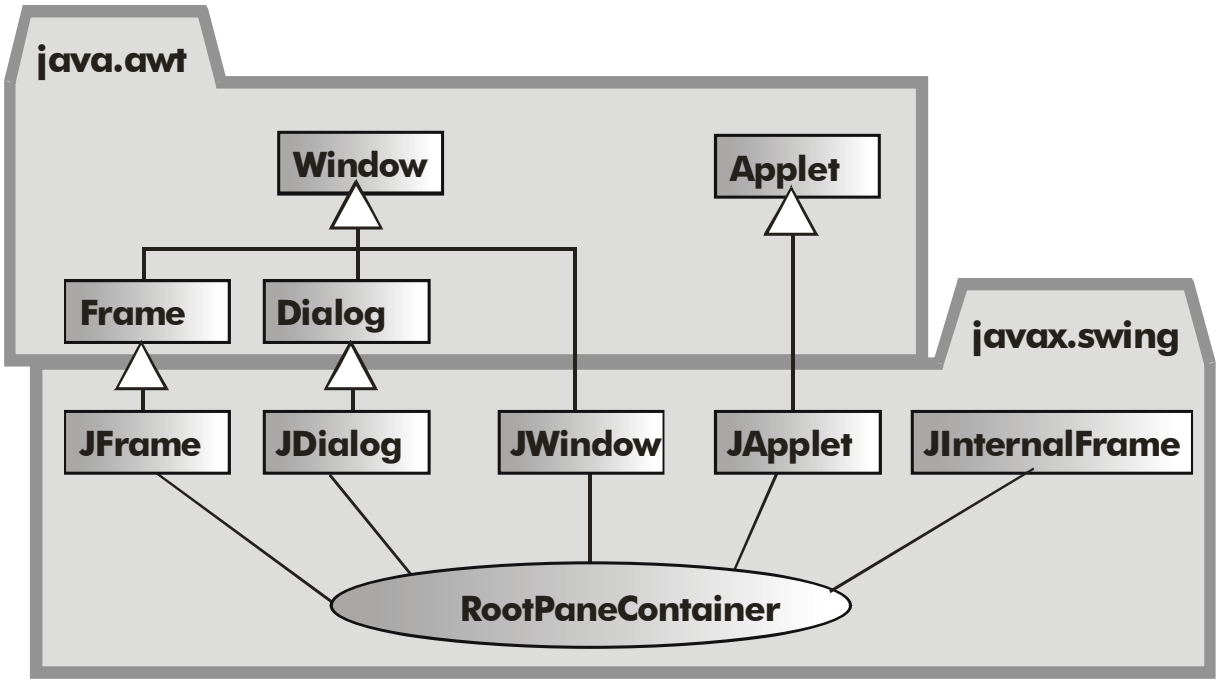


Ilustración 22, Marcos contenedores que implementan RootPaneContainer

Esta interfaz (*RootPaneContainer*) hace que las clases que la implementan utilicen un panel raíz para los contenidos llamado **JRootPane**. Es el nombre de una clase que implementa una estructura de paneles a todos los marcos ya comentados. La estructura que propone es la indica en las siguientes ilustraciones:

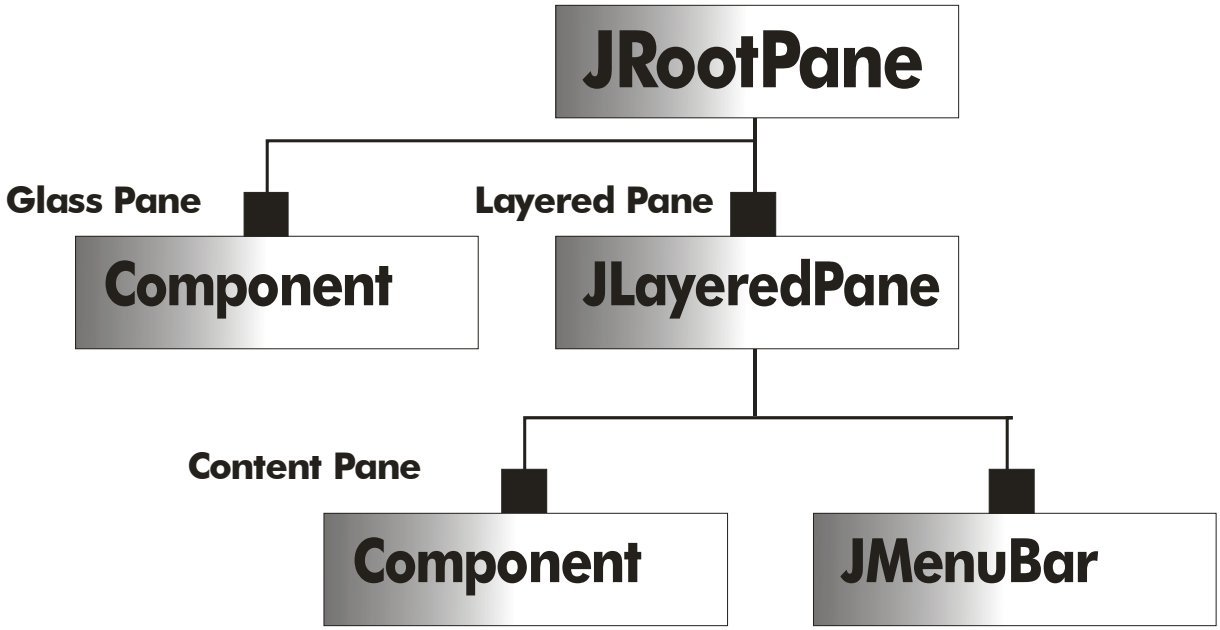


Ilustración 23, Estructura de los paneles de los marcos contenedores

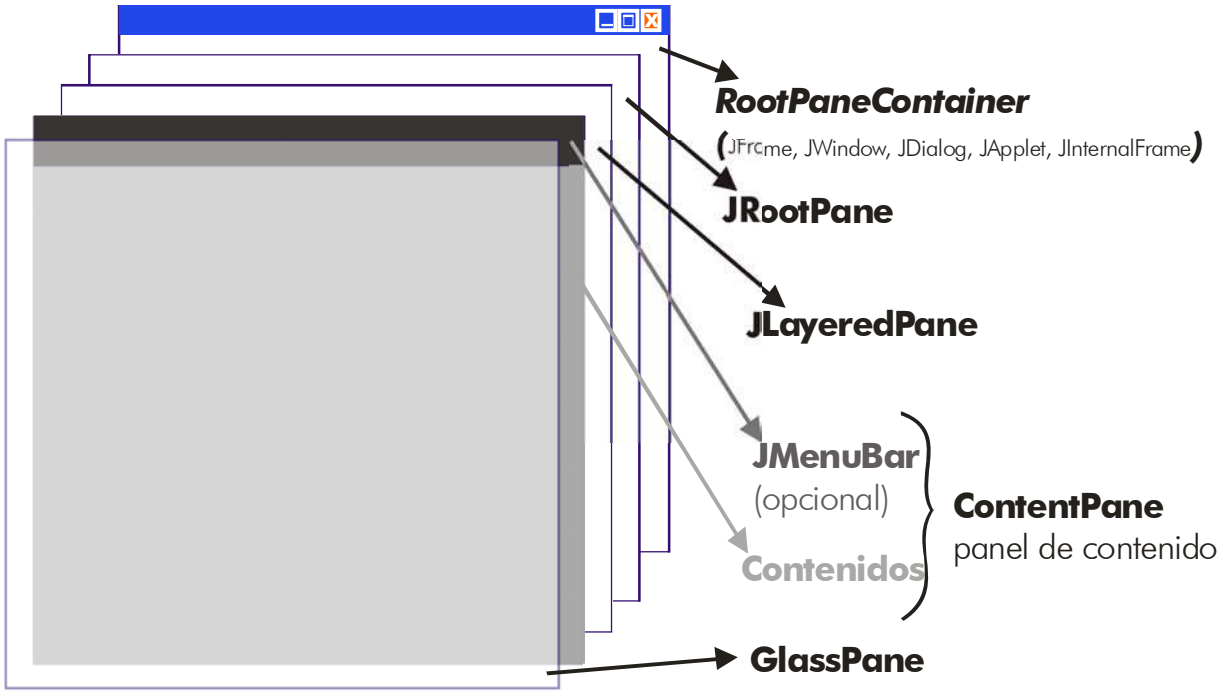


Ilustración 24, Apariencia visual de los paneles de los marcos contenedores

En la última ilustración, se puede observar la estructura de los paneles. De la que cabe destacar lo siguiente:

- La propia ventana se encuentra al fondo.
- Por encima se sitúa el **JRootPane** que está compuesto de dos grandes paneles:

- El **glass pane**, panel de cristal que queda encima de todo lo demás (virtualmente sería el *cristal de la pantalla*)
- El **panel de capas** (**JLayeredPane**) en el que se sitúan todos los demás componentes.

- El panel de capas está compuesto por el **panel de contenidos** (**content pane**) y el panel de barras de menú. Además dispone de cinco capas en las que colocar los componentes.
- El panel de contenidos es en el que se sitúan normalmente los contenidos de una ventana.

root pane. panel raíz

La clase **JRootPane** es la encargada de gestionar la apariencia de los objetos *JApplet, JWindow, JDialog, JInternalFrame* y *JFrame*. Se trata de un objeto hijo de estas clases. La clase *JRootPane* deriva de **JComponent.**

El *root pane* (panel raíz) de los componentes JFrame y JApplet posee dos objetos: son el *glass pane* y el *layered pane,* como ya se comentó anteriormente.

El panel raíz se puede obtener mediante el método **getRootPane()**, de las clases contenedoras.

layered pane. panel de capas

Este panel está compuesto por el *content pane* y las barras de menú. Divide el rango en varias capas (descritas de la más profunda a la más cercana):

- **JLayeredPane.DEFAULT_LAYER.** Capa en la que se colocan los componentes. Es la capa estándar y la que está más abajo.
- **JLayeredPane.PALETTE_LAYER**. Capa de paletas. Pensada para barras de herramientas y paletas flotantes.
- **JLayeredPane.MODAL_LAYER.** Cuadros de diálogo modales.
- **JLayeredPane.POPUP_LAYER**. Capa para el menú emergente.
- **JLayeredPane.DRAG_LAYER**. Para realizar arrastre con el ratón.

Estas cinco capas se pueden utilizar ya que se corresponden a propiedades estáticas de la clase **JLayeredPane**. Además se corresponden a un número entero que va subiendo de 100 en 100, es decir la DEFAULT_LAYER está asociada a l 0, la PALETTE_LAYER al 100, etc. Una pega es que como propiedades devuelven un objeto **Integer** por lo que hay que usar el método **intValue** de esta clase para convertirlas a entero normal (**int**).

Para obtener el panel de capas se usa el método **getLayeredPane,** devuelve un objeto **JLayeredPane.** Este panel se utiliza sólo para realizar gráficos en los que se simula una profundidad de capas. La forma de posicionar elementos por capas consiste en utilizar el método **setLayer**. Mediante este método se colocar un componente en una determinada capa y en una posición concreta en la capa (esto último es opcional, y se indica con un número entero que no debe de pasar de 100). **moveToFront** coloca un componente arriba de la capa, **moveToBack** le coloca detrás y **setPosition** en una posición concreta. Ejemplo:

```
//en un JFrame por ejemplo
JButton b=new JButton("Estoy muy arriba");
getLayeredPane().setLayer(b,
        JLayeredPane.POPUP_LAYER.intValue());
getLayeredPane().add(b);//El botón está más alto de lo normal
```

glass pane. panel de cristal

Está colocado encima de los demás paneles, en general es el panel en el que se realizan los movimientos de ratón. Se puede dibujar en él, pero no es recomendable.

content pane. panel de contenido

Esa es la capa en la que se dibujan los componentes normales. Es un objeto *Container* (al igual que el *glass pane*).

El método **getContentPane** devuelve el *content pane* de la ventana, cuadro de diálogo o applet actual. El método **add**, permite añadir componentes al *content pane*. Los componentes se deberían colocar en este panel.

clase JPanel

Esta clase es la clase básica contenedora de Swing. Se puede preparar para dibujar componentes y elementos, y luego asignarla por ejemplo al *content pane* de una aplicación mediante el método **add**.

Ejemplo:

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Container;
import java.awt.Color;

public class pruebaJPanel {
   public static void main(String args[]){
        //Se crea un objeto JPAnel de 300X300 y color rojo
       JPanel panelRojo =new JPanel();
       panelRojo.setBackground(Color.RED);
       panelRojo.setSize(300,300);

        //Se crea una ventana de 300X300
       JFrame ventana=new JFrame("Prueba en rojo");
       ventana.setLocation(100,100);
       ventana.setSize(300,300);
       ventana.setVisible(true);
```

```
//Se coloca el JPanel en el content pane
        Container contentPane=ventana.getContentPane();
        contentPane.add(panelRojo);
    }
}
```

El resultado anterior es una ventana de 300 por 300 píxeles con fondo de color rojo. Lo bueno es que como JPanel es un *Container* se le pueden añadir componentes con el método **add.**

Normalmente se suele generar una clase que deriva de JPanel y en ella redefinir el método **paintComponent** para definir la forma de dibujar objetos en el panel.

constructores

constructor	descripción
JPanel()	Construye un nuevo JPanel con doble búfer y composición *flow layout*
JPanel(boolean dobleBúfer**)**	Crea un JPanel e indica si se desea un doble búfer para él y composición *flow layout*
JPanel(LayoutManager l**)**	Construye un nuevo JPanel con doble búfer y la composición marcada por *l*
JPanel(boolean dobleBúfer, LayoutManager l**)**	Construye un panel indicando si se desea doble búfer y el tipo de composición

métodos

método	descripción
PanelUI **getUI()**	Devuelve el tipo de UI que se está usando en forma de objeto *PanelUI*
String **getUIClassID()**	Devuelve el tipo de UI que se está usando en forma de cadena
void **setUI(*PanelUI* ui)**	Asigna un nuevo UI al panel
void **updateUI()**	Hace que el panel acepte los cambios de UI realizados normalmente con **UIManager.setLookAndFeel**

clases de dibujo y contextos gráficos

Una de las tareas actualmente más requerida en la programación es la de generar aplicaciones que incluyan imágenes y dibujos. Esto pertenece al contexto de la informática gráfica. Hoy en día es casi indispensable su uso ya que pertenecemos a un mundo cada vez más visual y menos textual.

En el caso de Java hay diversos paquetes que se encargan de apoyar este uso de la programación. Son:

- **java.awt.color** Para representar colores (ya comentada anteriormente en este manual, véase 133).
- **java.awt.font.** Para representar tipos de letra.

- **java.awt.geom.** Para representar formas geométricas. Contiene las clases de dibujo de formas (**Arc2D, QuadCurve2D**, etc.)
- **java.awt.image.** Para representar imágenes
- **java.awt.print.** Para conectar con la impresora
- **java.awt.Graphics2D.** Es la nueva clase que representa contextos gráficos. Sustituye a la anterior **java.awt.Graphics**

De todas ellas la más importante es **Graphics2D** que es la clase perteneciente al nuevo soporte gráfico Java 2D. Mejora sustancialmente los gráficos existentes en Java hasta la versión 1.1

Esta clase representa el contexto en el que se dibujan los gráficos, y suministra métodos avanzados para crear gráficos en pantalla. Para conseguir el objeto *Graphics2D* en el que deseamos dibujar se pueden utilizar estos métodos:

- Utilizar desde clases AWT o Swing como resultado de una petición de dibujo. Esta petición se pasa al método **paint**, **paintComponent** o **update** que reciben un objeto **Graphics** que mediante *cast* puede convertirse a **Graphics2D** (ya que la clase Graphics2D es hija de Graphics).
- Pidiéndole de modo directo a un búfer de memoria
- Copiando de modo directo un objeto *Graphics2D* ya existente

método *paint* y *paintComponent*

paint es el método heredado de **JComponent** encargado de dibujar los elementos del componente, de los componentes ineriores, uso de la memoria intermedia para gráficos, preparación del contexto gráfico, ... Además como cada componente posee este método, el paint del objeto contenedor llama los paint correspondientes del resto de componentes.

En el caso de los componentes es recomendable usar **paintComponent** para dibujar los gráficos (en detrimento de **paint**). La razón es que este método dibuja el contenido sólo del componente, además el método **paint** de la clase contenedora se encarga de llamar a estos métodos de las clases hijas**,** por lo que éstas no deben utilizar paint.

Es más paint delegada su trabajo en otros tres métodos **paintComponent** (encargado de redibujar el componente), **paintBorder** (para redibujar el borde) y **paintChildren** (encargada de repintar los componentes contenidos en el actual). Pero se sigue utilizando el método *paint* para indicar como se debe dibujar el área de trabajo.

Nunca se ha de llamar a **paint** o **paintComponent** explícitamente, sino que se llama automáticamente por parte del panel siempre que haga falta o bien se utiliza la función **repaint** en el caso de que se necesite su llamada explícita (no se debe usar **update** como ocurría en AWT).

normas para pintar con Swing

En los programas Swing, a la hora de escribir el código de pintado, hay que entender perfectamente las siguientes reglas.

1> Para los componentes Swing, *paint()* se invoca siempre como resultado tanto de las órdenes del sistema como de la aplicación; el método *update()* no se invoca jamás sobre componentes Swing

2> Jamás se debe utilizar una llamada a *paint*, este método es llamado automáticamente. Los programas pueden provocar una llamada futura a *paint()* invocando a *repaint()*.

3> En componentes con salida gráfica compleja, *repaint()* debería ser invocado con argumentos para definir el rectángulo que necesita actualización, en lugar de invocar la versión sin argumentos, que hace que sea repintado el componente completo

4> La implementación que hace Swing de *paint()*, reparte esa llamada en tres llamadas separadas: *paintComponent()*, *paintBorder()* y *paintChildren()*. Los componentes internos Swing que deseen implementar su propio código de repintado, deberían colocar ese código dentro del ámbito del método *paintComponent()*; ***no*** dentro de *paint()*. **paint** se utiliza sólo en componentes grandes (como JFrame).

5> Es aconsejable que los componentes (botones, etiquetas,...) estén en contenedores distintos de las áreas de pintado de gráficos para evitar problemas. Es decir se suelen fabricar paneles distintos para los gráficos que para los componentes.

representación de gráficos con Java 2D

Gracias a este conjunto de paquetes se puede:

- Producir una forma
- Rellenar con colores sólidos o con texturas
- Transformar los objetos
- Recortar formas para restringirlas a un área
- Establecer reglas de composición para indicar como se combinan los píxeles que se quieren pintar sobre píxeles previos ya pintados
- Indicar modos de representación para acelerar la representación de gráficos (a costa de la velocidad).

Los pasos detallados para crear gráficos en Java 2D son:

1> Obtener un objeto *Graphics2D* una forma común de hacerlo es a través del método **paint** o **paintComponent**.

```
public void paintComponent(Graphics g){
   Graphics2D g2=(Graphics2D) g;
...
```

2> Establecer los consejos (*hints*) de representación gráficos

3> Establecer el trazo de dibujo mediante **setStroke** que poseen los objetos **Stroke:**

```
Stroke trazo=....;
g2.setStroke(trazo);
```

4> Establecer la pintura (indica cómo se rellenan las formas, color, textura, etc.) creando objetos **Paint** y utilizando el método **setPaint**

```
Paint pintura=...;
g2.setPaint(pintura);
```

5> Usar objetos de recorte (clase **Shape**) para delimitar la imagen a través del método **setClip.**

```
Shape forma=...;
g2.clip(forma);
```

6> Transformar el dibujo mediante métodos de transformación o una transformación afín

```
g2.setTransform(AffineTransform.getRotationInstance(0.9);
```

7> Transformar desde el espacio de usuario hasta el espacio de dispositivo si es necesario realizar conversión de coordenadas (sólo es necesario si se desea trabajar con coordenadas personales, independientes de dispositivo).

8> Establecer reglas de composición para indicar como se mezclan los nuevos píxeles con los ya existentes.

```
Composite mezcla=...;
g2.setComposite(mezcla);
```

9> Crear la forma combinando y mezclando los dibujos anteriores:

```
Shape forma=....
```

10> Dibujar y rellenar la forma

```
g2.draw(forma);
g2.fill(forma);
```

Casi nunca se realizan todos los pasos.

formas 2D

En Java 1.0 la clase *Graphics* definía métodos para dibujar formas. Desde 1.2, es decir desde la aparición de Java 2D, se utilizan clases que representan estas formas. Todas las

clases implementan la interfaz **Shape** que define unos cuantos métodos relacionados con formas gráficas bidimensionales.

métodos de la interfaz Shape

Como se ha dicho anteriormente, las formas 2D implementan esta interfaz. Con lo cual los métodos definidos en ella sirven para todas las formas. Son:

método	descripción
boolean contains(double x, **double** y**)**	Chequea si el punto x, y está en el interior de la forma, si es así devuelve **true.**
boolean contains(Point2D p**)**	Chequea si el punto p (de tipo **java.awt.geom.Point2D)** está en el interior de la forma, si es así devuelve **true.**
boolean contains(double x, **double** y, **double** ancho, **double** alto**)**	Devuelve **true** si el rectángulo formado por la coordenada x,y (referida a la esquina superior izquierda del rectángulo) y la anchura y altura indicadas, están totalmente dentro de la forma.
boolean contains(Rectangle2D r**)**	Devuelve **true** si el rectángulo r (de tipo **java.awt.geom.Rectangle2D)** está en el interior de la forma
Rectangle getBounds()	Devuelve un objeto **java.awt.Rectangle** que contiene totalmente a la forma
Rectangle2D getBounds2D()	Devuelve un objeto **java.awt.Rectangle2D** que contiene totalmente a la forma
PathIterator getPathIterator(AffineTransform at**)**	Obtiene un objeto iterador que interacciona con la superficie de la forma y permite acceso a su geometría.
boolean intersects(double x, **double** y, **double** ancho, **double** alto**)**	Devuelve **true** si el rectángulo formado por la coordenada x,y (referida a la esquina superior izquierda del rectángulo) y la anchura y altura indicadas, hace intersección con la forma
boolean intersects(Rectangle2D r**)**	Devuelve **true** si el rectángulo r hace intersección con la forma

formas rectangulares

La superclase **RectangularShape** es la clase padre de **Rectangle2D** (que representa rectángulos), **RoundRectangle** (rectángulo de esquinas redondeadas), **Ellipse2d** (elipses) y **Arc2D** (Arcos).

Todas las formas descritas se puede entender que están inscritas en un rectángulo, lo que facilita su uso. De hecho, todas derivan de la clase **RectangularShape** que proporciona métodos interesantes para manipular las formas. Son (además de los heredados de la clase Shape):

constructor	uso
double getCenterX()	Devuelve la coordenada X del centro del rectángulo

constructor	uso
double getCenterY()	Devuelve la coordenada Y del centro del rectángulo
double getHeight()	Altura del rectángulo
double getMaxX()	Coordenada X más larga del rectángulo
double getMaxY()	Coordenada Y más larga del rectángulo
double getMinX()	Coordenada X más corta del rectángulo
double getMinY()	Coordenada Y más corta del rectángulo
double getWidth()	Anchura del rectángulo
double getX()	Coordenada X de la esquina superior izquierda del rectángulo
double getY()	Coordenada Y de la esquina superior izquierda del rectángulo
void setFrame(double x, **double** y, **double** anchura, **double** altura**)**	Establece nueva medida y posición para el rectángulo
void setFrameFromCenter(double x, **double** y, **double** esquinaX, **double** esquinaY**)**	Modifica el rectángulo para que obtenga su posición a partir del centro marcado por x, y por las dos coordenadas de una de las esquinas
void setFrameFromDiagonal(double x1, **double** y1, **double** x2, **double** y2**)**	Modifica el rectángulo para que obtenga su posición a partir de las coordenadas de los dos vértices indicados

Las coordenadas de todas las clases que terminan en 2D se pueden obtener en formato **float** o **double**, a través de una clase inscrita que poseen todas ellas (**Rectangle2D.Float** y **Rectangle2D.Double** por ejemplo). La ventaja de *float* es que proporcionan suficiente precisión gastando menos memoria. La ventaja de *double* es que se manipula más rápido por parte del procesador y da mayor precisión. Sólo en la construcción hay que elegir este detalle

Creación:

```
Rectangle2D rectangulo=new Rectangle2D.Float(5F,10F,7.5F,15F);
Rectangle2D rect2=new Rectangle2D.Double(5,10,7.5,15);
```

constructores

Sólo se comenta la versión **Double** pero hay que recordar que hay versión **Float** (que sería igual pero cambiando los tipos **double** por **float**)

constructor	uso
Rectangle2D.Double(double x, **double** y, **double** ancho, **double** alto**)**	Crea un rectángulo con la esquina superior izquierda en la coordenada *x,y* y una anchura y altura dadas
Ellipse2D.Double(double x, **double** y, **double** ancho, **double** alto**)**	Crea una elipse inscrita en el rectángulo definido por esos cuatro parámetros

constructor	uso
Arc2D.Double(double x, double y, double ancho, double alto**,** double inicio, double longitud, int tipoDeCierre**)**	Crea un arco inscrito en el rectángulo definido por los cuatro primeros parámetros. El arco se empezará a dibujar desde el *inicio* marcado en ángulos (0, 90, etc.) y durante los grados marcados por *longitud* (inicio 90 y longitud 90, significa un ángulo de 90 a 180 grados). *E*l tipo de cierre puede ser alguna de estas constantes: ◉ Arc2D.**OPEN.** El arco no se cierra ◉ Arc2D.**CHORD.** El arco se cierra con una recta ◉ Arc2D**.PIE.** El arco se cierra en forma de tarta, las líneas van de los extremos de la curva al centro de la elipse.
Arc2D.Double(Rectangle2D rectángulo**,** double inicio, double longitud, int tipoDeCierre**)**	Igual que el anterior sólo que los cuatro primeros parámetros se indican con un objeto en forma de rectángulo
RoundRectangle2D.Double (double x, double y, double ancho, double alto**,** double anchoEsquina, double alturaEsquina**)**	Crea un rectángulo de esquinas redondeadas. Los cuatro primeros parámetros indican la forma del rectángulo sin redondear. Las dos últimas indican el tamaño del arco de las esquinas.

puntos

La clase **Point2D** crea puntos en el espacio que se pueden aplicar a diversas funciones. Su construcción requiere también elegir entre su clase interna **Float** o **Double**. Ejemplo:

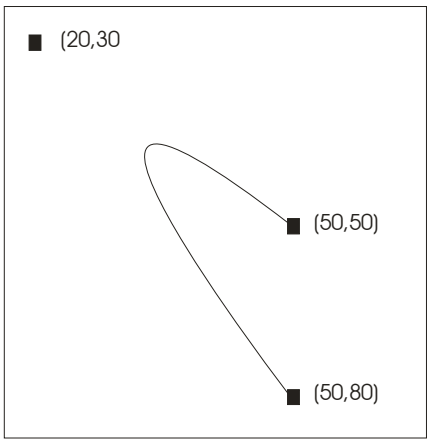
```
Point2D p=new Point2D.Double(10,45);
```

curvas

Java 2D proporciona curvas cuadráticas y cúbicas. Las cuadráticas son complejas de manipular para crearlas se utilizan dos coordenadas iniciales y dos finales más dos coordenadas de control de curva. Ejemplo:

```
QuadCurve2D c=new QuadCurve2D.Double(50,50,20,30,50,80)
```

El resultado es:



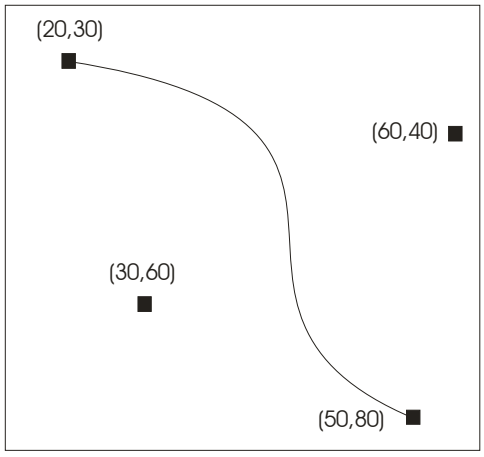
Las cúbicas representan **curvas de Bezier** que permiten dibujar curvas más eficientemente a través de puntos de tangente. El formato es:

```
new CubicCurve2D.Double(inicioX, inicioY, controlInicioX,
controlInicioY, controlFinX, controlFinY, finX, finY);
```

Ejemplo:

```
CubicCurve2D c=new
CubicCurve2D.Double(20,30,30,60,60,40,50,80)
```

El resultado es:

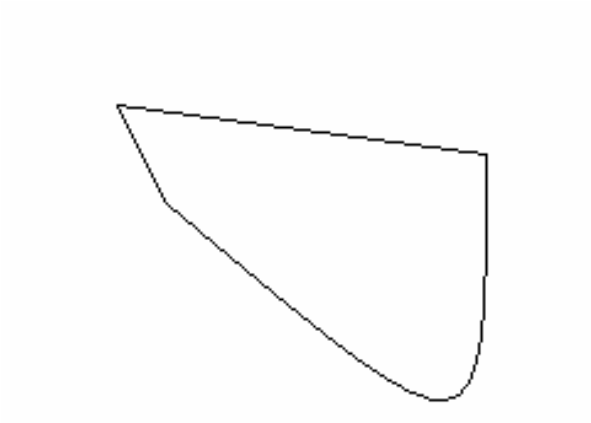


Hay una forma mucho más compleja que es el objeto **GeneralPath.** Este objeto permite encadenar curvas y rectas. Se construye con el constructor sin argumentos o utilizando un constructor que usa como argumento una forma ya hecha que será la primera curva de la secuencia.

Después el método **moveTo** se especifica la primera coordenada. El método **lineTo** permite dibujar una línea desde el punto anterior a uno nuevo. El método **curveTo** permite dibujar una curva de Bezier al siguiente punto. Los parámetros son:

```
curveTo(controlPuntoAnteriorX,
controlPuntoAnteriorY,controlX, controlY, X, Y);
```

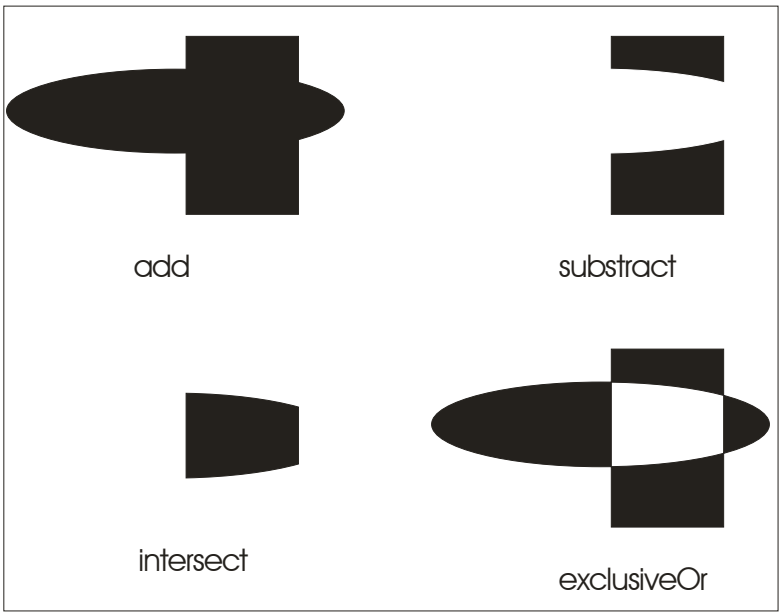
El método **closePath** permite cerrar la curva. Ejemplo:



El método **moveTo** no sirve sólo para empezar la ruta, marca también otro segmento de ruta.

áreas

Una técnica muy interesante para construir formas consiste en usar operaciones lógicas para unir, restar o intersectar figuras. Hay cuatro operaciones de este tipo:



Para realizar estas operaciones se requiere un objeto de área (clase **Area**). Es esta clase la que posee estos métodos. La clase **Area** posee un constructor en el que se pueden

colocar objetos **Shape**. De esta forma se pueden combinar figuras como se muestra en el ejemplo:

```
Graphics2D g2=(Graphics2D) g;
Area a1=new Area(new Rectangle2D.Double(100, 100, 400,200));
Area a2=new Area(new Ellipse2D.Double(150,50, 250, 400));
a1.add(a2);
g2.draw(a1);
```

trazos

Define los bordes de las formas dibujadas. Por defecto es una línea de un píxel que contornea a la forma. La interfaz **Stroke** es el que permite definir trazos. En Java2D sólo la clase **BasicStroke** (dentro de **java.awt**)implementa dicho interfaz. Esta interfaz devuelve sólo un método llamado **createdStrokedShape** que recibe una forma **Shape** y devuelve otra forma cuyo contenido es el trazo definido alrededor del **Shape** utilizado como argumento. El método **setStroke** de la clase **Graphics2D** es el que establece el trazo actual. Ejemplo:

```
//si g2 es un Graphics2D
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Ellipse2D.Double(15,40,140,310));
```

En el ejemplo la elipse tendrá una anchura de 10 píxeles.

constructores de *BasicStroke*

constructor	uso
BasicStroke(float ancho)	Crea un trazo con la anchura dada
BasicStroke(float ancho, int finalLínea, int unión)	Crea un trazo con la anchura dada. Especifica también un final de línea que puede ser: ◉ BasicStroke**.CAP_BUTT.** Final de línea en recto ◉ BasicStroke**.CAP_ROUND.** Final de línea redondeada ◉ BasicStroke**.CAP_SQUARE.** Final de línea en cuadrado (el borde se extiende más que en recto), Se especifica también la forma de esquinas de las líneas ◉ BasicStroke**.JOIN_BEVEL.** Vértice de línea en bisel ◉ BasicStroke**.JOIN_MITER.** Vértice de línea en inglete ◉ BasicStroke**.JOIN_ROUND.** Vértice de línea en redondeado

constructor	uso
BasicStroke(float ancho, int finalLínea, int unión, float límite**,** float[] trazoLínea, float faseDeTrazo)	Igual que la anterior pero con otros tres parámetros más: el máximo desfase de las esquinas y extremos de la línea, un array de números *float* para indicar el grosor de los trazos de la línea y un último parámetro para indicar en que posición comienzan esos trazos.

Ejemplo:

```
float lineaPuntoYRaya={20,5,5,5};//Raya (20), espacio (5)
                                 //punto(5), espacio(5)
g2.setStroke(new BasicStroke(15F, BasicStroke.CAP_BUTT,
BasicStroke.JOIN_MITER, 10F, lineaPuntoYRaya,0);
```

Dibuja línea de 15 píxeles en recto con forma de línea en punto y raya.

pintura

La interfaz **Paint** permite indicar de qué forma se rellenarán las formas dibujadas. El método **setPaint** de la clase **Graphics2D** es el que permite indicar la forma de la pintura. Este método requiere una clase que implemente la interfaz **Paint.**

Tras utilizar el método **setPaint,** los métodos **fill** (pintar el relleno) y **draw** (pintar el contorno) rellenan con el color elegido.

pintar con color

La clase Color tratada anteriormente implementa el interfaz **Paint**, por lo que el relleno con un color es muy sencillo. Ejemplo:

```
g2.setPaint(Color.RED);
```

Esta instrucción coloca rojo como el color de pintura. Este será el color que utilice el método **fill** que es el que realmente rellena una forma (que se debe pasar como argumento) con la pintura elegida. Si tras **setPaint**, el método que se usa es **draw**, entonces es el borde el que aparecerá de color rojo.

pintar con gradientes

Un gradiente es una forma de pintura en degradado desde un color a otro. Se crea mediante la clase **GradientPaint** a la que se la pasan dos puntos y el color de cada uno.

constructores de *GradientPaint*

constructor	uso
GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2**)**	Gradiente que va desde la coordenada *(x1,y1)* a la coordenada (*x2,y2*) y que hace transición de colores (gradiente) desde el *color1* hasta el *color2*
GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cíclico**)**	Igual que el anterior, sólo que permite que el gradiente sea cíclico (se repite continuamente)
GradientPaint(Point2D p1, Color color1, Point2D p1, Color color2**)**	Versiones de los constructores anteriores usando objetos *Point2D* para especificar las coordenadas
GradientPaint(Point2D p1, Color color1, Point2D p1, boolean cíclico**)**	

Ejemplo:

```
Point2D p1=new Point2D.Double(80,100);
Point2D p2=new Point2D.Double(200,200);
g2.setPaint(new
  GradientPaint(p1,Color.BLACK,p2,Color.YELLOW));
g2.fill(a1);
g2.draw(a1);
```

Y esto es equivalente a:

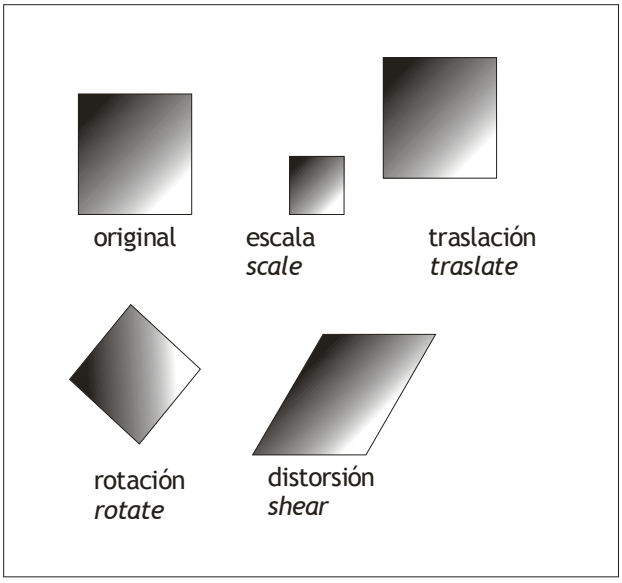
```
g2.setPaint(new GradientPaint(80,100,Color.BLACK,
200,200,Color.YELLOW);
g2.fill(a1);
```

Otra posibilidad es pintar utilizando una textura, como se verá más adelante.

transformaciones

Son operaciones que se realizan sobre el contexto gráfico, de forma que los siguientes dibujos aparecen distorsionados en la dirección marcada por la transformación. Lo que se transforma es el contexto gráfico.

Esta transformación se ejecuta con algunos de estos métodos:



Son los métodos **scale, translate, rotate** y **shear** los encargados de realizarlas. Se pueden realizar todas las transformaciones deseadas.

método de *Graphics2D*	uso
void rotate(double ángulo**)**	Rota el contexto el ángulo. indicado. El ángulo indica radianes.
void rotate(double ángulo**, double** x, **double** y**)**	Rota desde el punto *x,y*
void scale(double escalaX, **double** escalaY**)**	Escala la figura según los valores indicados
void shear(double x, **double** y**)**	Distorsiona la figura, según los ejes dados
void translate(double x, **double** y**)**	Mueve la figura

Hay que tener en cuenta que lo que se gira es el contexto gráfico, es decir, se gira el contexto y se dibuja la forma.

transformaciones afines

La clase **AffineTransform** permite realizar todo tipo de transformaciones. Se basa en el uso de una matriz de transformaciones. Y permite ejecutar todo tipo de operaciones sobre el contexto gráfico.

El método **setTransform** de la clase *Graphics2D* permite aplicar una transformación afín sobre el contexto gráfico para que al dibujar el siguiente objeto se utilice.

Una posibilidad rápida es usar los métodos estáticos de esta clase que permiten rotar, escalar, distorsionar, etc. Ejemplo:

```
g2.setTransform(AffineTransform.getRotateInstance(45));
g2.draw(rectangulo);
```

El rectángulo se dibuja rotado 45 grados. Habrá que devolver el contexto gráfico a su posición inicial para dibujar normal. El método **getTransform** de la clase *Graphics2D* permite obtener la transformación actual que se aplica sobre el contexto gráfico.

Métodos de AffineTransform:

método	uso
static AffineTransform getRotateInstance(double theta**)**	Rota el contexto el número de radianes indicado por *theta*
static AffineTransform getRotateInstance(double theta, **double** x, **double** y**)**	Rota el contexto el número de radianes indicado por *theta* girando desde el centro marcado por x,y
static AffineTransform getScaleInstance(double escalaX, **double** escala**)**	Escala el contexto según los valores de escala indicados
static AffineTransform getShearInstance(double escalaX, **double** escala**)**	Distorsiona la figura según los valores indicados
static AffineTransform getTranslateInstance(double x, **double** y**)**	Mueve el contexto según los valores indicados de desplazamiento.

recorte

Permite recortar la siguiente forma a dibujar. Tres métodos de *Graphics2D* permiten realizar estos recortes:

- **void setClip(Shape** s**).** Establece la forma *s* como forma de recorte del contexto gráfico. Lo siguiente que se dibuje quedará recortado según esta forma.
- **Shape getClip().** Devuelve la forma de recorte actual.
- **void clip(Shape** s**).** Más recomendable que la primera. Establece un área de recorte temporal para dibujar el siguiente elemento.

Así para recortar se hace:

```
Shape clipAntiguo=g2.getClip();
g2.clip(formaDeRecorte);
g2.setClip(clipAntiguo);//Se recupera el recorte inicial
```

composición

La composición permite indicar la manera en la que los dibujos de formas se mezclan. Es decir, cuando se dibuja un objeto en el contexto gráfico (**fuente**), los colores de éste se mezclan con los ya existentes (**destino)**. Sin indicar nada, el objeto fuente reemplaza a los destinos.

Para ello se utilizan objetos de tipo **AlphaComposite** que permiten indicar un grado de transparencia y una forma de mezcla. Estos objetos se crean a través del método estático **getInstance**.

Este método posee dos versiones, en la primera se establece la **regla** de fusión (un entero que se usa a través de las constantes especificadas en las siguientes tablas). La segunda versión añade un número **float** que es un valor entre 0 y 1, que indica el porcentaje. Un alpha 1 significa que la imagen se escribe al 100% de tinta, 0% es absolutamente transparente. Las constantes estáticas para la regla son:

constante	uso
AlphaComposite.CLEAR	Se eliminan los píxeles del objeto destino y del fuente.
AlphaComposite.DST	Dibuja sólo el objeto destino
AlphaComposite.DST_ATOP	Dibuja el objeto fuente, pero en la parte donde se solapa con el destino dibuja el destino
AlphaComposite.DST_IN	Se dibuja la parte del destino que se superpone al fuente.
AlphaComposite.DST_OUT	Se dibuja la parte del destino que no se solapa al fuente
AlphaComposite.DST_OVER	Se dibuja la parte del destino que se superpone al fuente.
AlphaComposite.SRC	Dibuja el color y el valor de transparencia del objeto fuente
AlphaComposite.SRC_ATOP	Dibuja el destino, pero en las partes en las que se solapan, dibuja el fuente
AlphaComposite.SRC_IN	Dibuja la parte del objeto fuente que se solapa con el destino
AlphaComposite.SRC_OUT	Dibuja la parte del objeto fuente que no se solapa con el destino
AlphaComposite.SRC_OVER	Dibuja el objeto fuente sobre el objeto de destino
AlphaComposite.XOR	Dibuja el objeto fuente y el destino en aquellas zonas donde no hay solapamiento

El método **setComposite** de la clase **Graphics2D** acepta un objeto de este tipo que afectará a lo siguiente que se dibuje.

fuentes

Se puede dibujar también texto en los contextos gráficos. El texto queda determinado fundamentalmente por su tamaño y tipo de letra. respecto al tipo de letra conviene utilizar tipos genéricos: **Dialog, DialogInput, Monospaced, Serif,** o **SansSerif**.

Suele haber otras cincuenta o sesenta fuentes más. Hay una clase llamada **GraphicsEnvironment** que se refiere a las propiedades del entorno gráfico en ejecución. Posee un método estático llamado **getLocalGraphicsEnvironment** que devuelve un objeto del mismo tipo referido al entorno local. Este objeto permite saca múltiples propiedades sobre las posibilidades gráficas de la máquina en ejecución, entre ellas **getAvailableFontFamilyNames,** devuelve los tipos de letra disponibles:

```
String lista[]=GraphicsEnvironment
               .getLocalGraphicsEnvironment()
               .getAvailableFontFamilyNames();
for(int i=0;i<lista.length;i++){
   System.out.println(lista[i]);
}
```

Lógicamente utilizar esas fuentes es un riesgo ya que no estarán presentes en todos los sistemas; de ahí que sea más recomendable el uso de fuentes genéricas.

clase Font

La clase **Font** sirve para establecer una determinada fuente. Se crea una fuente de esta forma:

```
Font f=new Font("SansSerif",Font.BOLD, 12);
```

Crea una fuente *SansSerif* de tamaño 12 puntos y negrita (*Font.ITALIC* es cursiva; la negrita y cursiva se coloca con *Font.ITALIC | Font.BOLD*)

Se pueden crear fuentes utilizando fuentes **TrueType** (en la que se basa la tipografía de Windows) haciendo uso de este listado (aunque sólo desde la versión 1.3 de Java):

```
URL u=new URL("http://www.fuentes.com/Pinoccio.ttf");
InputStream in=u.openStream();
Font f=Font.createFont(Font.TRUETYPE_FONT, in);
//Obtiene la fuente con tamaño 1, para subirla el tamaño
f=f.deriveFont(12.0F);//12 puntos
```

En el futuro puede que se soporten otros formatos de fuente (como por ejemplo **PostScript Tipo 1**).

Para dibujar texto en el panel de dibujo, se debe usar el objeto **Graphics** correspondiente. Este objeto posee un método **setPaint** que permite utilizar un objeto de fuente **Font** concreto. El método **drawString** es el encargado de dibujar el texto, lo hace a partir de unas coordenadas que indicarán la posición de la línea base a la izquierda del texto:

```
public void paint(Graphics g)
   Graphics2D g2=(Graphics2D) g;
...
   Font f=new Font("Dialog",Font.NORMAL,12);
   g2.setFont(f);
   g2.drawString("Hola",30,200);
```

métodos de font

método	uso
boolean canDisplay(char c)	Indica si la fuente posee el símbolo perteneciente al carácter indicado
int canDisplayUpTo(String s)	Indica si la fuente puede mostrar los caracteres integrados en la cadena *s*. De no ser así devuelve la posición dentro de la cadena del primer carácter que no se puede mostrar. Si puede mostrar toda la cadena, devuelve -1
static Font createFont(int formatoFuente, **InputStream** stream**)**	Obtiene una fuente a partir de un archivo de fuentes (ver ejemplo más arriba)
Font deriveFont(AffinTransform at**)**	Crea un nuevo objeto **Font** duplicando la fuente actual y transformándola según las características del objeto de transformación indicado.
Font deriveFont(float tamaño**)**	Devuelve un nuevo objeto **Font** resultado de ampliar o reducir el tamaño de la fuente. Para ampliar tamaño debe ser mayor que uno; para reducir debe ser menor que uno
Font deriveFont(int estilo**)**	Obtiene un objeto **Font** resultado de añadir el estilo indicado (FONT.Bold, FONT.Italic, ...) en la fuente.
Font deriveFont(Afint estilo**, float** tamaño**)**	Aplica el estilo y el tamaño indicado en una nueva fuente creada a partir de la actual.
byte getBaselineFor(char c)	Devuelve la línea de base apropiada para esta fuente
String getFamily()	Devuelve el nombre de familia de la fuente
String getFamily(Locale l**)**	Devuelve el nombre de familia de la fuente, para la configuración local indicada
String getFontName()	Devuelve el nombre de la fuente
String getFontName (Locale l**)**	Devuelve el nombre de la fuente, para la configuración local indicada
float getItalicAngle()	Devuelve el ángulo que utiliza la letra en cursiva
LineMetrics getLineMetrics(String s, **FontRenderContext** frc)	Obtiene un objeto de tipo **LineMetrics** configurado para la cadena s y el objeto de relleno de fuentes *frc*. Los objetos LineMetrics permiten obtener todas las medidas de dibujo de una fuente (descendente, interlineado)
String getName()	Devuelve el nombre lógico de la fuente
String getPSName()	Devuelve el nombre PostScript de la fuente
int getSize()	Devuelve el tamaño en puntos de la fuente

método	uso
float **getSize2D()**	Devuelve el tamaño en puntos de la fuente, ahora en formato **float**
Rectangle2D **getStringBounds**(String s, FontRenderContext frc)	Devuelve la forma rectangular mínima que contiene los caracteres de la cadena s utilizando el objeto **FontRenderContext** indicado (sirve para indicar como pintar las fuentes)
int **getStyle()**	Devuelve el estilo de la fuente (Font.BOLD, Font.ITALIC, ...)
boolean **isBold()**	Devuelve **true** si la fuente está en negrita
boolean **isItalic()**	Devuelve **true** si la fuente está en cursiva
String **toString()**	Devuelve el texto de la fuente

LineMetrics

La clase LineMetrics (en **java.awt.font**) se utiliza para obtener información sobre las medidas de una fuente en concreto. Se utiliza para dibujar elementos que se ajusten correctamente sobre un texto concreto. La forma de hacerlo es:

```
public void paint(Graphics g)
   Graphics2D g2=(Graphics2D) g;
...
   FontRenderContext frc=g2.getRenderContext();
   LineMetrics lm=font.getLineMetrics("Abuelo",frc);
```

El objeto *frc* es un **FontRenderContext** que se utiliza para saber cómo se pintan las fuentes en el contexto gráfico.

Ilustración 25, medidas de las fuentes. LineMetrics

A partir del objeto LineMetrics se pueden obtener estos métodos:

método	uso
float **getAscent()**	Medida del ascendente de la línea
float **getBaselineIndex()**	Devuelve el tipo de línea base del texto. Puede ser una de estas constantes: ROMAN_BASELINE, CENTER_BASELINE, HANGING_BASELINE.

método	uso
float **getBaselineOffsets()**	Devuelve el desplazamiento de las letras a partir de la línea base
float **getDescent()**	Medida del descendente de la línea
float **getHeight()**	Altura del texto
float **getLeading()**	Medida del interlineado
float **getNumChars()**	Número de caracteres máximos a los que este objeto puede incluir
float **getStrikeThroughOffset()**	Posición de la línea de tachado a partir de la línea base del texto
float **getStrikeThroughThickness()**	Grosor de la línea de tachado
float **getUnderlineOffset()**	Posición de la línea de subrayado a partir de la línea base del texto
float **getUnderlineThickness()**	Grosor de la línea de subrayado

clase TextLayout

Es una potentísima clase (dentro de **java.awt.font**) que permite usar texto como si fuera una forma, lo que permite operaciones muy avanzadas.

Se crean objetos de este tipo mediante un String que representa el texto, un objeto *Font* con las características de la fuente y un objeto **FontRenderContext.** Esta última clase sirve para saber como dibujar el texto. Se puede obtener un objeto de este tipo con el método **getFontRenderContext** de la clase **Graphics2D**. Ejemplo de creación:

```
FontRenderContext frc=g2.getFontRenderContext();
TextLayout tl=new TextLayout("Dialog", new Font("SansSerif",
                        Font.BOLD, 12),frc);
```

El método **draw** permite dibujar el texto en la zona deseada de pantalla. Para usar este método se debe pasar el objeto *Graphics2D* en el que deseamos dibujar y las coordenadas x e y que marcan la posición del texto.

Por otro lado esta clase posee numerosos métodos para realizar todo tipo de transformaciones sobre el texto. Por ejemplo **getOutline** permite obtener una forma *Shape* del texto aplicando además una determinada transformación (mediante un objeto **AffineTransform**).

métodos

método	uso
void **draw(**Graphics2D g2**,** float x**,** float y**)**	Dibuja el texto en el contexto gráfico indicado a partir de las coordenadas x, y
float **getAscent()**	Medida del ascendente de la línea
float **getDescent()**	Medida del descendiente de la línea

método	uso
float **getBaseline()**	Devuelve el tipo de línea base del texto. Puede ser una de estas constantes: **Font**.ROMAN_BASELINE, **Font**.CENTER_BASELINE, **Font**.HANGING_BASELINE.
float **getBaselineOffsets()**	Devuelve el desplazamiento de las letras a partir de la línea base
Rectangle2D **getBounds()**	Obtiene un rectángulo delimitador que engloba al texto
int **getCharacterCount()**	Número de caracteres del texto
float **getLeading()**	Obtiene la posición de la cabecera del texto
Shape **getOutline(**AffineTransform trans**)**	Obtiene la forma del contorno del texto usando la transformación indicada. La forma posee coordenadas en 0,0. Teniendo en cuenta que en el caso del texto la referencia se toma sobre la línea base, con lo que la forma del texto se dibujará fuera de la pantalla si no se transforma.

Además posee numerosos métodos (aquí no comentados) para obtener posiciones de texto en el caso de que se desee intercalar algo al actual

imágenes de mapas de bits

Todas las imágenes diseñadas en este capítulo son imágenes vectoriales. No obstante Java tiene capacidad para manejar imágenes GIF, JPEG o PNG. También puede manipular vídeo y GIFs animados.

La clase **java.awt.Image** es la encargada de representar imágenes. Para construir un objeto de este tipo se puede utilizar el método **getImage** disponible en las clases applet. Pero si queremos mostrar una imagen en otro componente, entonces debemos utilizar el llamado **Toolkit** que es una clase especial que posee métodos muy interesantes. Este *toolkit* se obtiene utilizando el método **getToolkit** disponible en todos los componentes.

Por ello, para obtener una imagen se usa:

```
Image imagen=getToolkit().getImage(url);
```

La ***url*** es la dirección URL a la imagen. Si queremos obtener la imagen de nuestra carpeta de recursos. Se puede utilizar:

```
Image imagen=getToolkit().getImage(getClass().getResource
                                        ("dibujo1.gif"));
```

Tras estas funciones, Java no habrá cargado la imagen. Se cargará cuando se intente mostrar. Hay una interfaz llamada **ImageObserver** que sirve para examinar la imagen a cargar y así conocer sus condiciones a priori (anchura, altura, tamaño, etc.).

Las imágenes se suelen dibujar usando la función **drawImage()** definida en la clase *Graphics2D*. Esta función permite dibujar una imagen usando un ImageObserver. Por suerte, todos los componentes implementan la interfaz *ImageObserver*.

En su forma más básica **drawImage** dibuja imágenes usando la esquina de la imagen, el nombre de un objeto **Image** y un **ImageObserver.** Ejemplo:

```
g2.drawImage(image,50,50,this);
```

método	uso
boolean drawImage(Image imagen, **int** x, **int** y, **ImageObserver** observador**)**	Dibuja la imagen en la posición *x, y* usando el observador indicado
boolean drawImage(Image imagen, **int** x, **int** y, **Color** fondo, **ImageObserver** observador**)**	Dibuja la imagen en la posición *x, y* usando el observador indicado y el color de fondo que se indica
boolean drawImage(Image imagen, **int** x, **int** y, **int** ancho, **int** alto, **Color** fondo, **ImageObserver** observador**)**	Dibuja la imagen en la posición x, y con la altura y anchura indicada y el Color de fondo señalado.
boolean drawImage(Image imagen, **int** x, **int** y, **int** ancho, **int** alto, **ImageObserver** observador**)**	Dibuja la imagen en la posición x, y con la altura y anchura indicada
boolean drawImage(Image imagen, **int** x1, **int** y1, **int** x2, **int** y2, **ImageObserver** observador**)**	Dibuja la imagen en el rectángulo definido por las coordenadas *x1,y1* y *x2,y2* Ajustando la imagen si es necesario.
boolean drawImage(Image imagen, **int** x1, **int** y1, **int** x2, **int** y2, ***Color*** fondo, ***ImageObserver*** observador**)**	Dibuja la imagen en el rectángulo definido por las coordenadas *x1,y1* y *x2,y2* Ajustando la imagen si es necesario.
boolean drawImage(Image imagen, **int** x1, **int** y1, **int** x2, **int** y2, **int** ox1, **int** oy1, **int** ox2, **int** oy2, ***ImageObserver*** observador**)**	Dibuja una imagen en el rectángulo definido por las coordenadas *x1,y1* y *x2,y2* La imagen a dibujar se toma del archivo de origen usando el rectángulo en esa imagen que va de *ox1, oy1* a *ox2,oy2* La imagen se transformará si es necesario.
boolean drawImage(Image imagen, **int** x1, **int** y1, **int** x2, **int** y2, **int** ox1, **int** oy1, **int** ox2, **int** oy2, ***Color*** fondo, ***ImageObserver*** observador**)**	Igual que el anterior pero usando un color de fondo para los píxeles transparentes.

Los métodos de la clase *Image,* **getHeight** y **getWidth** obtienen la altura y la anchura respectivamente de la imagen. Si no se conoce aún este dato, devuelven -1.

Threads

Introducción

En informática, se conoce como **multitarea**, la posibilidad de que una computadora realice varias tareas a la vez. En realidad es una impresión (salvo en un equipo con varios procesadores) que se consigue repartiendo tiempo y recursos entre distintos procesos.

La palabra *thread* hace referencia a un flujo de control dentro de un programa (también se le llama **hilo**). La capacidad que permiten los threads a un programa estriba en que se pueden ejecutar más de un hilo a la vez.

Los hilos comparten los datos del programa (además pueden tener datos propios) y esto hace que el control sea más dificultoso. Como ejemplo de *thread*, está el recolector de basura de Java que elimina los datos no deseados mientras el programa continúa con su ejecución normal.

El uso de hilos es muy variado: animación, creación de servidores, tareas de segundo plano, programación paralela,...

clase *Thread* y la interfaz *Runnable*

La interfaz **java.lang.Runnable** permite definir las operaciones que realiza cada *thread*. Esta interfaz se define con un solo método público llamado **run** que puede contener cualquier código. y que será el código que se ejecutará cuando se lance el *thread*. De este modo para que una clase realiza tareas concurrentes, basta con implementar *Runnable* y programar el método **run.**

La clase **Thread** crea objetos cuyo código se ejecute en un hilo aparte. Permite iniciar, controlar y detener hilos de programa. Un nuevo *thread* se crea con un nuevo objeto de la clase **java.lang.Thread**. Para lanzar hilos se utiliza esta clase a la que se la pasa el objeto **Runnable.**

Cada clase definida con la interfaz **Runnable** es un posible objetivo de un thread. El código de **run** será lo que el thread ejecute.

La clase Thread implementa el interfaz Runnable, con lo que si creamos clases heredadas, estamos obligados a implementar *run*. La construcción de objetos Thread típica se realiza mediante un constructor que recibe un objeto Runnable.

```
hilo = new Thread(objetoRunnable);
hilo.start(); //Se ejcuta el método run del objeto Runnable
```

métodos de Thread

métodos	uso
void **interrupt ()**	Solicita la interrupción del hilo
static boolean **interrupted()**	**true** si se ha solicitado interrumpir el hilo actual de programa
static void **sleep(int** milisegundos**)**	Duerme el Thread actual durante un cierto número de milisegundos.

métodos	uso
static void **sleep(int** milisegundos, **int** nanos**)**	Duerme el Thread actual durante un cierto número de milisegundos y nanosegundos.
boolean **isAlive()**	Devuelve verdadero si el thread está vivo
boolean **isInterrupted()**	**true** si se ha pedido interrumpir el hilo
static Thread **currentThread()**	Obtiene un objeto Thread que representa al hilo actual
void **setDaemon(**boolean b**)**	Establece (en caso de que *b* sea verdadero) el Thread como servidor. Un thread servidor puede pasar servicios a otros hilos. Cuando sólo quedan hilos de este tipo, el programa finaliza
void **setPriority(**int prioridad**)**	Establece el nivel de prioridad del Thread. Estos niveles son del 1 al 10. Se pueden utilizar estas constantes también: ◉ **Thread.NORMAL_PRIORITY.** Prioridad normal (5). ◉ **Thread.MAX_PRIORITY.** Prioridad alta (10). ◉ **Thread.MIN_PRIORITY.** Prioridad mínima (1).
void **start()**	Lanza la ejecución de este hilo, para ello ejecuta el código del método **run** asociado al *Thread*
static void **yield()**	Hace que el Thread actual deje ejecutarse a Threads con niveles menores o iguales al actual.

creación de *threads*

mediante clases que implementan Runnable

Consiste en:

1> Crear una clase que implemente **Runnable**

2> Definir el método **run** y en él las acciones que tomará el hilo de programa

3> Crear un objeto **Thread** tomando como parámetro un objeto de la clase anterior

4> Ejecutar el método **start** del **Thread** para ejecutar el método **run**

El código de *run* se ejecuta hasta que es parado el Thread. La clase Thread dispone de el método **stop()** para definitivamente la ejecución del thread. Sin embargo no es recomendable su utilización ya que puede frenar inadecuadamente la ejecución del hilo de programa. De hecho este método se considera obsoleto y **no debe utilizarse jamás.** La interrupción de un hilo de programa se tratará más adelante. Ejemplo de creación de hilos:

```
class animacionContinua implements Runnable{
  ...
  public void run() {
        while (true) {
              //código de la animación
              ...
        }
  }
...
animacionContinua a1 = new animacionContinua();
Thread thread1 = new Thread(a1);
thread1.start();
```

Se considera una solución más adaptada al uso de objetos en Java hacer que la propia clase que implementa **Runnable,** lance el **Thread** en el propio constructor.

```
class animacionContinua implements Runnable {
  Thread thread1;
  animacionContinua() {
        thread1 = new Thread(this);
        thread1.start();
  }
  ...
}
```

Es decir se lanza al thread en el propio constructor de la clase runnable.

mediante clases derivadas de Thread

Es la opción más utilizada. Como la clase **Thread** implementa la interfaz **Runnable**, ya cuenta con el método **run.** Pasos:

1> Crear una clase basada en **Thread**

2> Definir el método **run**

3> Crear un objeto de la clase

4> Ejecutar el método **start**

Ejemplo**:**

```
class animacionContinua extends Thread {
  public void run() {
        while (true) {
              //código de la animación
              ...
        }
  }
}
...
animacionContinua a1=new animacionContinua();
a1.start();
```

control de Threads

sleep

Para conseguir que un thread se pare durante un cierto tiempo sin consumir CPU, se utiliza o el método **sleep** de los objetos thread o el método estático **Thread.sleep.** En ambos casos se indica como parámetro el número de milisegundos que se detendrá la ejecución del thread. Se dice que el Thread **duerme** durante ese tiempo. Tras consumir el tiempo continuará la ejecución.

Este método detiene el hilo actual de programa, sea el que sea. Este método se debe realizar de vez en cuando en un hilo de ejecución continua ya que, de otro modo, se paraliza la ejecución del resto de Threads.

Es decir, dormir un Thread es obligatorio para la programación multitarea, mientras un hilo duerme, los otros se pueden ejecutar.

interrupción de la ejecución

Un detalle importante en el control de Threads es que los hilos no tienen ningún método de interrupción. Un *thread* finaliza cuando se abandona el método *run*. Pero, por supuesto, se debe dar la posibilidad de finalizar un *thread* y, por ello existe un método llamado **interrupt** que está relacionado con este hecho.

El método *interrupt* de la clase Thread, no interrumpe un hilo, pero sí solicita su finalización. Esta solicitud es realizada (se interrumpe el hilo) si el hilo está en ejecución, pero no si está dormido (con el método **sleep** o con el método **wait**). Cuando un hilo está dormido, lo que ocurre es una excepción del tipo **InterruptedException**.

Es decir si está dormido se produce una excepción, si el hilo está despierto queda marcado para su interrupción (el método **interrupted**, detecta si el hilo debe ser interrumpido). Visto así el cuerpo de un método **run** para que pueda ser parado, sería:

```
public void run(){
  try{
        while(condición de fin de bucle){
        // Instrucciones del hilo de programa
```

```
        }
   }catch(InterruptedException ie){
        //Se paro el hilo mientras estaba dormido
   }
   finalize{
        //Instrucciones de limpieza, si procede
   }
}
```

Lo malo es si la finalización ocurre cuando el Thread estaba despierto. En ese caso, según el ejemplo, no se saldría jamás. Por ello se utiliza el método estático booleano **interrupted** que vale **true** cuando se generó una interrupción del Thread durante su ejecución. El método **run** queda entonces así:

```
public void run(){
   try{
        while(!Thread.interrupted() && condición de fin de
                                        bucle){
        // Instrucciones del hilo de programa
        }
   }catch(InterruptedException ie){
        //Se paro el hilo mientras estaba dormido
   }
```

estados de un *thread*

Los hilos de programa pueden estar en diferentes estados. Cada uno de ellos permite una serie de tareas.

estado nuevo

Es el estado en el que se encuentra un *thread* en cuanto se crea. En ese estado el *thread* no se está ejecutando. En ese estado sólo se ha ejecutado el código del constructor del Thread.

estado de ejecución

Ocurre cuando se llama al método **start**. No tiene por qué estar ejecutándose el thread, eso ya depende del propio sistema operativo. Es muy conveniente que salga de ese estado a menudo (al estado de bloqueado), de otra forma se trataría de un hilo **egoísta** que impide la ejecución del resto de threads.

La otra posibilidad de abandonar este estado es debido a la *muerte* del thread

estado bloqueado

Un thread está bloqueado cuando:

- Se llamó al método **sleep**

- Se llamó a una operación de entrada o salida
- Se llamó al método **wait**
- Se intento bloquear otro thread que ya estaba bloqueado

Se abandona este estado y se vuelve al de *ejecutable* si:

- Se pasaron los milisegundos programados por el método **sleep**
- Se terminó la operación de entrada o salida que había bloqueado al thread
- Se llamó a **notify** tras haber usado **wait**
- Se liberó al thread de un bloqueo

estado muerto

Significa que el método finalizó. Esto puede ocurrir si:

- El flujo de programa salió del método **run**
- Por una excepción no capturada

Se puede comprobar si un thread no está muerto con el método **isAlive** que devuelve **true** si el thread no está muerto.

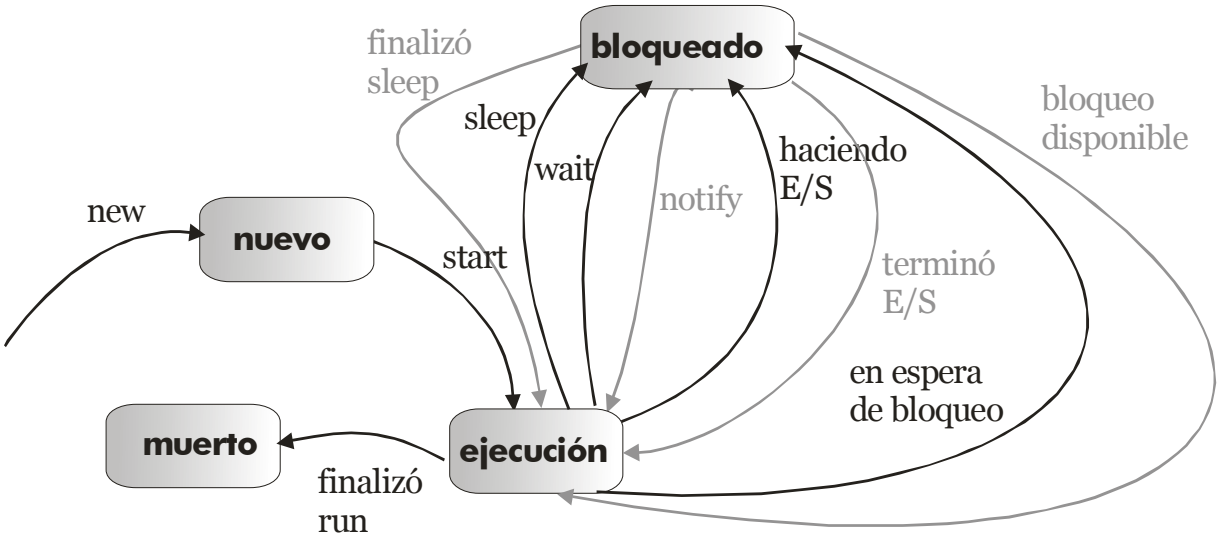


Ilustración 26, Diagrama de estados de un Thread o hilo

sincronización

Hay serios problemas cuando dos threads diferentes acceden a los mismos datos.

métodos sincronizados

Un primer problema ocurre cuando un método de un thread es interrumpido por otro thread de mayor prioridad. Por ejemplo: supongamos que poseemos un thread que usa un valor **x** y le multiplica por dos. Si desde que lee *x* hasta que la multiplica por dos, otro thread cambia el valor de x, entonces la multiplicación puede no ser correcta.

Para evitar que un método sea interrumpido por un thread externo, se utiliza la palabra clave **synchronized**:

```
public synchronized void multiplicar(){...
```

Estos métodos tienen como inconveniente, que consumen mucho tiempo de CPU, lo que aletargará mucho el programa si se utilizan muy a menudo.

método *wait*

Los métodos sincronizados impiden que otros threads les utilicen, por lo que bloquean a estos threads. Esto puede provocar que un programa quede paralizado si se está buscando una determinada condición que depende de otros threads.

Por ejemplo imaginemos esta situación: disponemos de threads que controlan cuentas de banco. Estos threads utilizan un método *sacarDinero* que sólo puede sacar dinero si efectivamente lo hay. Si este método está marcado con **synchronized** entonces el resto no pueden añadir dinero mientras se ejecute el anterior.

Esto lo soluciona el método **wait**. Este método bloquea al thread actual para que el resto pueda ejecutarse. Este método puede provocar una excepción del tipo **InterruptedException** si el thread es interrumpido durante la espera. Por eso los métodos *syncronized* deben propagar esta excepción a el thread que lo llamó mediante la palabra **throws** *(Véase* throws*, página 74)*. Ejemplo (método sacarDinero de una supuesta clase Banco, este método puede ser utilizado por un thread):

```
public class Banco...{
  ...
  public synchronized void sacarDinero(int Cantidad)
    throws InterruptedException{
       while (saldo<0){
            wait();//El thread se queda a la espera de que
                   //otro thread cambien el saldo
       }
  }
```

El método **wait** pertenece a la clase **Object** por lo que puede ser llamado por cualquier clase.

Al método *wait* se le pueden pasar un número máximo de milisegundos en espera, de modo que si la espera supera esos milisegundos, el thread se libera del bloqueo.

métodos *notify* y *notifyAll*

Los métodos bloqueados con **wait** se quedan en espera permanente y no salen de ella hasta que sean liberados por otros threads. Éstos, pueden liberar el bloqueo utilizando el método **notify** para liberar al thread bloqueado o el método **notifyAll** para liberar a todos los threads bloqueados.

Si no se llama a estos métodos, un thread en estado de espera, no se desbloquearía jamás. En general es mejor utilizar **notifyAll**, y éste debe ser llamado cuando el estado de un thread puede causar la ejecución de los threads en espera (en el ejemplo del banco habría que llamar a *notifyAll* cuando se utilice el método *ingresarDiner* por parte de un thread). Tanto **notify** como **notifyAll** son también métodos de la clase **Object.**

componentes Swing

introducción

Durante los temas anteriores se han tratado elementos de la construcción de interfaces de usuario. No obstante los paquetes de Java proporcionan clases especiales para crear interfaces de usuario. Además discutiremos también los principales apartados sobre la apariencia.

La colocación de componentes se realiza con el método **add** del contenedor en elque va el panel. En el caso de un elemento con **RootPane** (véase paneles de dibujo, página 160) como JFrame, JApplet y JDialog, se suele coolocar en el panel de contexto (que se obtiene con **getRootPane**).

administración de diseño

introducción

Los administradores de diseño son una parte esencial de la creación de interfaces de usuario, ya que determinan las posiciones de los controles en un contenedor. En lenguajes orientados a una sola plataforma, el problema es menor ya que el aspecto es más fácilmente controlable. Pero la filosofía Java está orientada a la portabilidad del código. Por eso este es uno de los apartados más complejos de la creación de interfaces, ya que las medidas y posiciones dependen de la máquina en concreto.

En otros entornos los componentes se colocan con coordenadas absolutas. En Java se desaconseja esa práctica porque en muchos casos es imposible prever el tamaño de un componente.

En su lugar se utilizan administradores de diseño que permiten realizar colocaciones y maquetaciones de forma independiente de las coordenadas.

El método **setLayout** dentro de la clase **Container** es el encargado de proporcionar un administrador de diseño a un determinado panel. Este método tiene como único parámetro un objeto de tipo **LayoutManager**. LayoutManager, es una interfaz implementada en diversas clases que sirven para maquetar (**FlowLayout, GridLayout, BoxLayout, BorderLayout, GridBagLayout, ...**)

La adición de elementos se puede crear con el método **add**. A veces no se muestran los cambios en el contenedor, para forzarles hay que utilizar el método **validate** que poseen los contenedores.

Flow Layout

Distribuye los componentes del contenedor de izquierda a derecha y de arriba abajo. Es la distribución más fácil y una de las más efectivas.

constructores

constructor	uso
FlowLayout()	Construye el administrador de tipo *flow* con alineación centrada y márgenes a 5 píxeles

constructor	uso
FlowLayout(int alineación**)**	Permite indicar la alineación que puede indicarse por alguna de estas constantes de clase: LEFT, RIGHT o CENTER
FlowLayout(int alineación, int sepH, int sepV**)**	Permite indicar alineación y la separación horizontal y vertical entre los componentes.

Grid Layout

Crea distribuciones en forma de malla que posee una serie de columnas y filas. Estas filas y columnas crean celdas de exactamente el mismo tamaño. Los componentes se distribuyen desde la primer celda a la izquierda de la primera fila; y van rellenando fila a fila toda la malla hasta la celda más a la derecha de la última fila.

constructores

constructor	uso
GridLayout()	Crea una malla de una sola celda
GridLayout(int nFilas, **int** nColumnas**)**	Crea una malla de las filas y columnas indicadas
GridLayout(int nFilas, int nColumnas, int sepH, int sepV**)**	Crea una malla con las filas y columnas indicadas y con los espacios entre botones que se especifican.

Ilustración 27,botones colocados en el panel de contenido con cuatro filas y cuatro columnas con separación vertical y horizontal entre componentes

El método **add** de los contenedores admite un segundo parámetro con el que se puede indicar el número de la celda donde se coloca el componente. Ejemplo:

```
getContentPane().add(boton15,5);
```

En el ejemplo el botón llamado *boton15* se coloca en la sexta casilla (la primera es la 0). Si había más casillas detrás de esa, entonces se mueven al siguiente hueco.

Si se añaden más componentes que casillas tenga el contenedor, entonces se amplía el Grid en consecuencia automáticamente.

Border Layout

Permite colocar componentes alrededor de los bordes de un contenedor. Por defecto es lo que utiliza AWT (y por lo tanto las clases Swing). Los bordes son **NORTH. SOUTH, EAST, WEST** y **CENTER.** Se suele utilizar estas formas en el método **add** para colocar componentes en el panel deseado:

```
JPanel jp=new JPanel();
jp.setLayout(new BorderLayout(2,3));//2 y 3 es el espaciado
jp.add(componente1,BorderLayout.NORTH);
jp.add("South",componente2);
```

Cualquiera de esas dos formas es válida. El panel central se come los paneles adyacentes si detecta que están vacíos.

constructores

BorderLayout()	Crea un nuevo *Border Layout*
BorderLayout(int espH, int espV**)**	Crea un nuevo *Border Layout* con los espacios señalados

BoxLayout

Permite distribuir componentes en una fila o columna. Pensado para filas y columnas de botones, pertenece al paquete **javax.swing** (las anteriores están en **java.awt**).

Para facilitar su manejo, Swing incluye un contenedor llamado Box que está pensado para manipular los componentes insertados en el contenedor. Box es una clase que posee diversos métodos estáticos que manipular internamente el administrador **BoxLayout**. Para crear un contenedor Box:

```
Box horizontal=Box.createHorizontalBox();
Boxvertical=Box.createVerticalBox();
```

Después se añaden componentes con el método **add** de la clase Box. Finalmente se coloca el componente Box dentro del panel / contenedor deseado.

métodos de la clase Box

constructor	uso
static Box **createHorizontalBox()**	Obtiene un contenedor Box horizontal para añadir componentes.

constructor	uso
static Box **createVerticalBox()**	Obtiene un contenedor Box vertical para añadir componentes.
static Component **createHorizontalGlue()**	Crea un componente horizontal para ajustar automáticamente la distancia horizontal entre los componentes. Devuelve el componente creado.
static Component **createHorizontalStrut(**int ancho**)**	Crea un componente horizontal con la anchura dada. Devuelve dicho componente.
static Component **createRigidArea(** Dimension d**)**	Crea un componente invisible con las dimensiones dadas.
static Component **createVerticalGlue()**	Crea un componente vertical para ajustar automáticamente la distancia vertical entre los componentes. Devuelve el componente creado.
static Component **createVerticalStrut(**int ancho**)**	Crea un componente vertical con la anchura dada. Devuelve dicho componente.

Ejemplo:

```
public static void main(String args[]){
   JFrame v=new JFrame();
   v.setLocation(50,50);
   v.setSize(400,300);
   v.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   JButton boton1=new JButton("Hola"),
         boton2=new JButton("Texto largo"),
         boton3=new JButton("Hola"),
         boton4=new JButton("Qué hay");
   Box b=Box.createHorizontalBox();
   b.add(boton1);
   b.add(b.createHorizontalGlue());
   b.add(boton2);
   b.add(b.createHorizontalGlue());
   b.add(boton3);
   b.add(b.createHorizontalGlue());
   b.add(boton4);
   v.getContentPane().add(b);
   v.setVisible(true);
}
```

El resultado es:

Aparecen separados los botones por los métodos **createHorizontalGlue.**

GridBagLayout

Es el administrador de diseño más flexible, pero es más complicado de manipular. Coloca los componentes en relación a sí mismos y gracias a ello se consigue (con paciencia y algo de suerte) cualquier diseño.

Los componentes se distribuyen en un espacio de coordenadas *lógicas*, que no reales, que son posiciones en el espacio de filas y columnas formadas por el grupo de componentes. De este modo tendremos una cuadrícula imaginaria que se estira formando tamaños basados en los componentes que se acomodan en ella.

Es mucho más cómodo trabajar con este diseño mediante programas de diseño WYSIWYG (como el editor de componentes de NetBeans).

Esta clase tiene un constructor sin parámetros y métodos para obtener información y configurar la malla de componentes creada.

GridBagConstraints

Esta es una clase totalmente asociada a este tipo de diseños, de hecho con esta clase se controla la posición y propiedades de los componentes añadidos a contenedores *GridBagLayout*. Posee una serie de propiedades

propiedades	uso
int **gridx, gridy**	Controla la posición del componente en la malla
int **weightx, weighty**	Indica como añadir el espacio sobrante a un componente
int **fill**	Controla la expansión del componente para llenar el espacio que sobre de su asignación
int **gridheight, gridwidth**	Controla el número de filas o columnas sobre las que se extiende el componente
int **anchor**	Posición del componente si hay espacio adicional sobrante
int **ipadx,ipady**	Relleno entre el componente y los bordes de su área
Insets **insets**	Configura el relleno de los bordes

colocación de componentes

En un **GridBagLayout** no se puede especificar el tamaño de la malla. Ésta se calcula según las necesidades. Esto se basa en lo siguiente: si un componente es añadido, por ejemplo, en la posición 25,25; entonces se crea una malla de esas dimensiones. Si el siguiente dice estar en la columna 30, la malla se amplía para que esto sea posible.

La colocación de componentes se basa en el método **add** de los contenedores. A este método se le pasa el componente a añadir y el conjunto de restricciones con que queda afectado. Esas restricciones se construyen con la clase *GridBagConstraints*. Las propiedades **gridx** y **gridy** permiten colocar al componente en una celda concreta de la malla.

Ejemplo:

```
public class VentanaGridBag extends JFrame {
   GridBagConstraints restricciones=new GridBagConstraints();
   public VentanaGridBag (){
         getContentPane().setLayout(new GridbagLayout());
         añadeGrid(new JButton("Norte"),1,0);
         añadeGrid(new JButton("Sur"),1,2);
         añadeGrid(new JButton("Este"),2,1);
         añadeGrid(new JButton("Oeste"),0,1);
         añadeGrid(new JButton("Centro"),1,0);
   }
   private void añadeGrid(Component c, int x, int y){
         restricciones.gridx=x;
         restricciones.gridy=y;
         getContentPane().add(c,restricciones);
   }
}
```

Resultado:

llenar los espacios

Se pueden expandir los controles a voluntad cuando no llenan el espacio completo de la celda asignada. Eso se realiza utilizando las propiedades **fill, weightx** y **weighty.**

fill permite indicar en que dirección se desplaza el componente para llenar el hueco. Se pueden utilizar las siguientes constantes estáticas:

- **GridBagConstraints.BOTH,** El componente se expande en horizontal y en vertical.
- **GridBagConstraints.HORIZONTAL,** El componente se expande en horizontal.
- **GridBagConstraints.VERTICAL,** El componente se expande en vertical.
- **GridBagConstraints.NONE,** El componente no se expande (opción por defecto)

Si en el ejemplo anterior añadimos, tras la línea del setLayout (subrayada en el ejemplo), esta línea:

```
        restricciones.fill.GridBagConstraints.BOTH
```

Todos los botones tendrán el mismo tamaño. Si además se añaden (debajo de la anterior):

```
        restricciones.weightx=1.0;
        restricciones.weighty=1.0;
```

Esto expande las celdas ocupadas, hasta llenar la ventana. El valor 0.0 no expande, cualquier valor distinto de 0, sí lo hace.

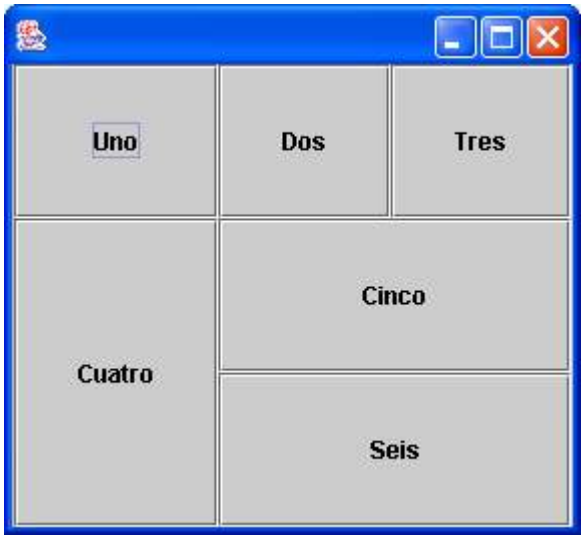
extensión de filas y columnas

La propiedad **gridheight** permite expandir un componente para que ocupe dos filas, **gridwidth** permite expandir un componente para que ocupe dos columnas. Ejemplo:

```
public class VentanaGridBag2 extends JFrame {
  GridBagConstraints restricciones=new GridBagConstraints();
  public VentanaGridBag2(){
        getContentPane().setLayout(new GridBagLayout());
        restricciones.fill=GridBagConstraints.BOTH;
        restricciones.weightx=1.0;
        restricciones.weighty=1.0;
        restricciones.gridwidth=2;
        añadeGrid(new JButton("Uno"),0,0);
                restricciones.gridwidth=1;
        añadeGrid(new JButton("Dos"),2,0);
                restricciones.gridheight=2;
        añadeGrid(new JButton("Tres"),0,1);
                restricciones.gridheight=1;
                restricciones.gridwidth=2;
```

```
        añadeGrid(new JButton("Cuatro"),1,1);
        añadeGrid(new JButton("Cinco"),1,2);
                restricciones.gridwidth=1;
    }
    private void añadeGrid(Component c, int x, int y){
        restricciones.gridx=x;
        restricciones.gridy=y;
        getContentPane().add(c,restricciones);
    }}
```

Resultado:



pesos

Sin duda es de los elementos más difíciles de controlar. Son las propiedades de *GridBagConstraints* **weigthx** y **weighty** las que permiten su control. Sólo valen de forma relativa. Es decir si a un componente le asignamos 0.5 y a otro 1.0, esto significa que el segundo es el doble de grande. El valor 0, significa que se mantendrá como estaba originalmente.

ubicación absoluta

Es una forma de trabajo que Sun desaconseja, aunque bien es cierto que nos libra de la tiranía de los administradores de diseño y nos permite una colocación libre, mediante píxeles. Es idónea cuando el tamaño en píxeles de la ventana es fijo.

Consiste en indicar **null** como valor para el método **setLayout** del contenedor en el que deseamos colocar nuestros componentes.

Tras esa acción, los componentes a colocar indicarán su posición y tamaño antes de añadirse al contenedor. Si no se desea calcular el tamaño, el método de la clase *Component* **getPreferredSize()** devuelve el tamaño predeterminado que se calculará automáticamente en función de su contenido.

Ejemplo:

```
public class pruebaLayoutAbsoluto {
    public static void main(String args[]){
        JButton boton=new JButton("Absoluto");
        JFrame v=new JFrame();
        v.setLocation(50,50);
        v.setSize(400,300);
        v.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        v.getContentPane().setLayout(null);
        boton.setLocation(230,90);
        boton.setSize(100, 80);
        v.getContentPane().add(boton);
        v.setVisible(true);
    }
}
```

Resultado:

apariencia

clase *UIManager*

Una de las principales mejoras de Swing fue el administrador de apariencia, conocido como **UIManager** y para el cual se creó la clase con el mismo nombre (integrada en **javax.swing**). Esta clase permite cambiar la apariencia según varios esquemas preestablecidos. La idea es que un programa Java se visualice igual independientemente de la plataforma.

Los tres esquemas son: **Metal** (que es el que funciona por defecto), **Motif** (parecida a la estética X-Windows) y **Windows** (parecida a la estética Windows).

Para cambiar la apariencia se utiliza el método estático (*UIManager* es una clase estática) **setLookAndFeel** al cual se le pasa una cadena con este texto según el formato deseado:

- ***javax.swing.plaf.metal.MetalLookAndFeel***

- ***com.sun.java.swing.plaf.motif.MotifLookAndFeel***
- ***com.sun.java.swing.plaf.windows.WindowsLookAndFeel***

El método *setLookAndFeel* lanza diversos tipos de eventos que hay que capturar, son: **ClassNotFoundException, InstantiationException, IllegalAccessException, UnsupportedLookAndFeelException.** Normalmente se capturan todos a la vez usando la superclase **Exception.**

Para obligar a un elemento a actualizar la apariencia se utiliza la función de las clases *JComponent*, **updateUI** sin argumentos. Ejemplo de cambio de apariencia:

```
import javax.swing.*;
public class pruebaTiposApariencia {
 public static void main(String args[]){
   JFrame ventana1=new JFrame();
   JPanel jp=new JPanel();
   jp.setSize(300,300);

   ventana1.setLocation(100,100);
   ventana1.setSize(300,300);
   jp.add(new JTextField("Texto de prueba"));
   jp.add(new JButton("Botón de prueba"));
   ventana1.setContentPane(jp);
   ventana1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   try{
         UIManager.setLookAndFeel("com.sun.java.swing.”+
         “plaf.windows.WindowsLookAndFeel");
   }
   catch(Exception e) {System.out.println(“Error”);
             System.out.println("");
         }
         jp.updateUI();
         ventana1.setVisible(true);
     }
```

Siempre hay una estética actual para la apariencia de la ventana. Es lo que se llama el ***LookAndFeel*** actual. LookAndFeel es una clase Swing preparada para gestionar apariencias. Se puede modificar esta clase estática para conseguir una apariencia personal (indicando tipos de letra, bordes, etc. etc.)

Para averiguar los sistemas **LookAndFeel** instalados se puede usar el método estático **UIManager.getInstalledLookAndFeels,** devuelve un array de objetos **UIManager.LookAndFeelInfo**. Esta clase posee un método **getClassName** que devuelve el nombre completo de los sistemas instalados.

etiquetas

Son textos de una línea que sirven para dar información textual a las ventanas y applets. También se pueden utilizar para mostrar imágenes estáticas.

creación de etiquetas

constructor	uso
JLabel()	Crea una etiqueta normal y sin imagen
JLabel(String texto**)**	Construye un objeto JLabel con el texto indicado
JLabel(String texto, **int** alineaciónHorizontal**))**	Construye un objeto JLabel con el texto y alineación indicados
JLabel(Icon imagen**)**	Construye un objeto JLabel con esa imagen
JLabel(Icon imagen, **int** alineaciónHorizontal**)**	Construye un objeto JLabel con esa imagen y alineación
JLabel(String texto, **Icon** imagen, **int** alineaciónHorizontal**)**	Construye un objeto JLabel con ese texto, imagen y alineación

Se pueden crear etiquetas de todo tipo, con texto e imágenes. La posición de la etiqueta dependerá del tipo de maquetación del contenedor (se verá más adelante). Un etiqueta normal se colocaría:

```
JFrame ventana1=new JFrame();
JPanel jp=new JPanel(false);
JLabel l1=new JLabel("Hola");
JLabel l2=new JLabel("Adiós");

jp.add(l1);
jp.add(l2);

ventana1.setLocation(100,100);
ventana1.setSize(300,300);
ventana1.setContentPane(jp);
ventana1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana1.setVisible(true);
```

Como se observa en el ejemplo, las etiquetas se pueden colocar con el método **add** de los paneles contenedores.

etiquetas HTML

En Java se permite que el texto de una etiqueta incluya comandos HTML. Basta utilizar estos comandos en el texto de la etiqueta. Se puede utilizar cualquier comando, pero sólo se aconseja usar comandos formato de texto y párrafo.

Para que el texto de una etiqueta se considere texto HTML y no texto normal, se debe comenzar el texto de la etiqueta con **<html>**.

Ejemplo:

```
JLabel textoHTML=new JLabel("<html>"+
  "<font color='Red' size='7' face='Arial,Helvetica,"+
  "sans-serif>Este es el texto</font>"
```

En el ejemplo anterior el texto *Este es el texto*, saldría de color rojo, tamaño 7 (el máximo) y letra Arial o Helvética o sans-serif.

etiquetas gráficas

Hay una modalidad de etiqueta que permite colocar imágenes. Las imágenes que se pueden colocar se basan en el interfaz **Icon**. Este interfaz define los siguientes métodos: **getIconHeight** (altura del icono), **getIconWidth** (anchura del icono) y **paintIcon** que pinta el icono dentro de componente especificado, indicando su apartado gráfico y sus posiciones x e y.

Para facilitar la tarea de usar iconos, se puede hacer uso de los objetos **ImageIcon**.

clase ImageIcon

Se trata de una clase pensada para utilizar iconos a partir de ficheros de imagen. Constructores:

constructor	uso
ImageIcon()	Crea un icono de imagen (vacío)
ImageIcon(String rutaArchivo**)**	Crea un icono de imagen a partir del archivo cuya ruta se indica
ImageIcon(String rutaArchivo, String descripción)	Crea un icono de imagen a partir del archivo cuya ruta se indica y utilizando la descripción especificada. La descripción sirve para indicar qué archivo se abre, no es obligatorio, es una formalidad.
ImageIcon(Image i**)**	Crea un icono imagen a partir del objeto de imagen *i*
ImageIcon(Image i, String descripción)	Crea un icono imagen a partir del objeto de imagen *i* utilizando la descripción especificada.
ImageIcono(URL url**)**	Crea un icono imagen desde la URL indicada
ImageIcon(URL url, String descripción**)**	Crea un icono imagen desde la URL indicada utilizando la descripción especificada.
ImageIcono(byte[] datos**)**	Crea un icono imagen desde un array de bytes
ImageIcon(byte[] datos, String descripción**)**	Crea un icono imagen utilizando la descripción especificada.

Las imágenes deben estar en formato GIF o JPG.

uso de etiquetas gráficas

basta utilizar un **ImageIcon** como parte de la construcción de la etiqueta. Si el icono de imagen existe y es correcto, se mostrará. Además se puede crear una etiqueta con texto e imagen:

```
JLabel grafica1=new JLabel(new ImageIcon("D:/hola.gif");
JLabel grafica2=new JLabel("Texto de la etiqueta",
                           new ImageIcon("D:/fondo.gif"));
```

En el segundo ejemplo se mostrará una imagen y un texto.

métodos para cambiar el texto y la imagen

método	uso
Icon getDisabledIcon()	Obtiene el icono que se muestra si la etiqueta está deshabilitada
Icon getIcon()	Obtiene el icono de la etiqueta (si lo hay)
String getText()	Obtiene el texto de la etiqueta
void setDisabledIcon(Icon i**)**	Asigna el icono especificado como imagen a mostrar si se deshabilita la etiqueta
void setIcon(Icon i**)**	Define la imagen que mostrará la etiqueta
void setText(String texto**)**	Asigna el texto especificado a la etiqueta
void setOpaque(boolean b**)**	Permite indicar si queremos que el fondo de la etiqueta tenga el texto opaco o no

alineación

método	uso
void setHorizontalAlignment(int alineación**)**	Fija la alineación horizontal de la etiqueta. Esta es la posición de la etiqueta (texto e imagen) con respecto a la horizontal del panel en el que está inscrita la etiqueta. Puede ser: ◉ **JLabel.LEFT** (izquierda) ◉ **JLabel.RIGHT** (derecha) ◉ **JLabel.CENTER** (centro)
void setVerticalAlignment(int alineación**)**	Fija la alineación vertical de la etiqueta con respecto al panel en el que está inscrita Puede ser: ◉ **JLabel.TOP** (arriba) ◉ **JLabel.BOTTOM**(abajo) ◉ **JLabel.CENTER** (centro)

método	uso
void setHorizontalTextPosition(int posición**)**	Fija la posición del texto respecto a la imagen de la etiqueta, referido a la horizontal. Las posibilidades son las mismas que con **setHorizontalAlignment**
void setVerticalTextPosition(int posición**)**	Igual que la anterior, pero referida a la vertical. Las opciones son las mismas que las de **setVerticalAlignment**

Todos estos métodos tienen versión **get** (**getVerticalAlignment**, por ejemplo) que permiten obtener las alineaciones actuales.

etiquetar otros componentes

Las etiquetas se pueden asignar a otros componentes, e incluso se las puede asignar una tecla asociada a un carácter mnemónico que permite dar el foco al componente asociado a la etiqueta. Un mnemónico es una inicial remarcada (en Windows, subrayada) que permite activar el componente usando Alt + letra remarcada. Los métodos relacionados son:

método	uso
Component **getLabelFor()**	Obtiene el componente que éste objeto etiqueta o **null** so no lo hay.
char getDisplayedMnemonic()	Obtiene el carácter fijado como mnemónico de la etiqueta.
void setLabelFor(*Component* c**)**	Hace que el componente *c* sea etiquetado por esta etiqueta
void setDisplayedMnemonic(char c**)**	Hace que el carácter *c* actúe como mnemónico de la etiqueta.
void setDisplayedMnemonicIndex(int i**)** throws IllegalArgumentException	Indica qué posición de letra es la que se remarcará (empezando por 0, que es la primera) Sirve para el caso en el que queramos resaltar una determinada letra como mnemónica y resulte que se repita varias veces. Lanza evento del tipo: **IllegalArgumentException** si esa posición no existe. Este método se incorporó en la versión 1.4

cuadros de texto

Son controles que permiten al usuario especificar un texto para que el programa lo examine. Es uno de los controles más usados.

constructores

constructor	uso
JTextField()	Crea un nuevo cuadro de texto con la configuración predeterminada
JTextField(int columnas**)**	Anchura del cuadro en caracteres
JTextField(String texto**)**	Crea un cuadro de texto conteniendo el texto que se pasa como argumento
JTextField(String texto, int columnas**)**	Crea un cuadro de texto conteniendo el texto que se pasa como argumento e indicando su anchura.

métodos

método	uso
int **getColumns()**	Devuelve la anchura actual del cuadro
void **setColumns(**int ancho**)**	Establece la anchura del cuadro
String **getSelectedText()**	Obtiene el texto seleccionado del cuadro
int **getSelectionStart()**	Devuelve la posición del texto del cuadro donde comienza el texto seleccionado
int **getSelectionEnd()**	Devuelve la posición del texto del cuadro donde finaliza el texto seleccionado
Color **getSelectionTextColor()**	Obtiene el color del texto seleccionado
Color **getSelectionColor()**	Obtiene el color de fondo del texto seleccionado
void **setSelectionStart(**int pos**)**	Hace que en la posición de texto indicada comience el texto seleccionado
void **setSelectionEnd(**int pos**)**	Hace que en la posición de texto indicada finalice el texto seleccionado
void **setSelectionTextColor(**Color color**)**	Asigna el color indicado como nuevo color del texto seleccionado del cuadro
void **setSelectionColor(**Color color**)**	Asigna el color indicado como nuevo color del fondo seleccionado del cuadro
void **setAlignmentX(**float alineación**)**	Establece la alineación horizontal del texto respecto al cuadro. Puede ser: ◉ **JTextField.LEFT_ALIGNMENT** (izquierda) ◉ **JTextField.RIGHT_ALIGNMENT** (derecha) ◉ **JTextField.CENTER_ALIGNMENT** (centro)

método	uso
void setAlignmentY(float alineación**)**	Establece la alineación vertical del texto respecto al cuadro. Puede ser: ◉ **JTextField.TOP_ALIGNMENT** (arriba) ◉ **JTextField.BOTTOM_ALIGNMENT** (abajo) ◉ **JTextField.CENTER_ALIGNMENT** (centro)
float getAlignmentX()	Obtiene la alineación horizontal del texto
float getAlignmentY()	Obtiene la alineación vertical del texto

eventos

Se producen eventos del tipo **ActionEvent** cuando se pulsa Intro en un cuadro de texto.

cuadros de contraseña

Se corresponden a la clase **JPasswordField** que es una subclase de **JTextField**, con lo que lo dicho para ésta vale para los cuadros de contraseña. La diferencia está en que en los cuadros de contraseña, el texto que se escribe queda oculto (normalmente por asteriscos) siendo ideal para introducir contraseñas en un programa.

métodos

Posee los mismos que los de **JTextField** ya que deriva de éste. Sin embargo están obsoletos los métodos **getText** y **setText:**

método	uso
char getEchoChar()	Obtiene el carácter que usa el cuadro de contraseña para ocultar el contenido.
char[] getPassword()	Obtiene el texto del cuadro de contraseña en forma de array de caracteres. Sustituye a **getText**
void setEchoChar(char c**)**	Configura el carácter *c* para que sea el carácter con el que se substituya el texto escrito en el cuadro

botones

Son elementos fundamentales en el trabajo de las aplicaciones. Cuando son pulsados, generan un evento *ActionEvent* que, capturado permite crear una acción para el botón. La clase fundamental es **JButton** que toma la mayor parte de su funcionalidad de **AbstractButton** clase abstracta con la que se definen casi todos los botones.

métodos interesantes de *AbstractButton* (válidos para todos los botones)

constructor	uso
void doClick()	Hace clic en el botón (igual que si el usuario lo hubiera hecho de verdad)
void doClick(int milisegundos)	Lo mismo, pero el clic se hace tras el número indicado de milisegundos.
void setIcon(Icon i**)**	Fijas el icono normal del botón
void setText(String texto**)**	Fija el texto del icono
void setDisabledIcon(Icon i**)**	Fija el icono para el estado de *desactivado* del botón
void setEnabled(boolean b**)**	Habilita y deshabilita al botón
void setPressedIcon(Icon i**)**	Establece el icono a mostrar cuando se hace clic sobre el cotón
void setRolloverIcon(Icon i**)**	Fija el icono que aparece cuando se arrima el ratón al botón.
void setRolloverEnabled(boolean b**)**	Indica si se permite o no efectos de *Rollover*
void setSelectedIcon(Icon i**)**	Fija el icono que aparecerá cuando el botón se selecciona
void setSelected(boolean b**)**	Hace que el botón esté o no seleccionado.
void setMnemonic(char c**)**	Indica qué letra hará de mnemónico del botón. Esa letra permitirá acceder al control desde el teclado.
void setHorizontalAlignment(int alineación**)**	Fija la alineación horizontal del botón. Puede ser: ◉ **JButton.LEFT** (izquierda) ◉ **JButton.RIGHT** (derecha) ◉ **JButton.CENTER** (centro)
void setVerticalAlignment(int alineación**)**	Fija la alineación vertical del botón. Puede ser: ◉ **JButton.TOP** (arriba) ◉ **JButton.BOTTOM**(abajo) ◉ **JButton.CENTER** (centro)
void setHorizontalTextPosition(int posición**)**	Fija la posición horizontal del texto del botón respecto a la imagen. Las posibilidades son las mismas que con **setHorizontalAlignment**
void setVerticalTextPosition(int posición**)**	Igual que la anterior, pero referida a la vertical. Las opciones son las mismas que las de **setVerticalAlignment**

constructores de JButton

constructor	uso
JButton()	Crea un nuevo botón

constructor	uso
JButton(String texto**)**	Crea un nuevo botón con el texto indicado
JButton(Icon icono**)**	Crea un nuevo botón con el icono indicado
JButton(String texto, Icon icono**)**	Crea un nuevo botón con el texto e icono indicado

botón por defecto

Cada ventana o cuadro de diálogo puede tener un botón por defecto. Este botón está asociado a la tecla Intro y aparece marcado de forma especial. Para hacer que un botón sea el botón por defecto:

```
JFrame ventana=new JFrame("Ventana1");
JButton boton1=new JButton("Aceptar");
...
ventana.getRootPane().setDefaultButton(boton1);
```

Es el método de **JRootPane** llamado **setDefaultButton** el que permite asignar un botón como botón por defecto.

eventos ActionEvent

Los botones y los cuadros de texto (y otros controles) generan eventos del tipo **ActionEvent**. Para manipular estos eventos, se debe llamar en cada control que queramos que lance eventos, al método **addActionListener** y se indicará el objeto que manipulará estos eventos (este objeto debe pertenecer a alguna clase que implemente el interfaz **ActionListener**). Será el método **actionPerformed** el que se encargará de manipular el evento.

Un problema típico consiste en que, a menudo, se necesita averiguar qué botón o cuadro de texto lanzó el evento. Una forma fácil de saberlo es mediante la cadena **ActionCommand.** Esta cadena es un texto que describe al objeto que lanzó el evento. Se usa de la siguiente forma:

1> El objeto que lanza eventos de este tipo rellena su cadena ActionCommand usando su método **setActionCommand**

2> En el método **actionPerformed** podremos leer esa cadena usando el método **getActionCommand** del evento ActionEvent que se usa como argumento.

Ejemplo:

```
JButton bt1=new JButton("Aceptar");
bt1.setActionCommand("bt1");//Se puede usar cualquier cadena
  ....
```

En el método **actionPerformed**, el código sería:

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

```
if (e.getActionCommand().equals("bt1")==true){
      System.out.println("Se pulsó Aceptar");
}
```

casillas de activación

Se trata de controles que permiten su activación y desactivación a fin de elegir una serie de opciones independientes. En Swing es la clase **JCheckBox** la encargada de representarlas. Esta clase deriva de **JToggleButton**, que, a su vez, deriva de **AbstractButton**

constructores

constructor	uso
JCheckBox()	Construye una nueva casilla de verificación
JCheckBox(String texto**)**	Crea una nueva casilla con el texto indicado
JCheckBox(Icon icon**)**	Crea una nueva casilla con el icono indicado
JCheckBox(String texto, **boolean** activado**)**	Crea una nueva casilla con el texto indicado y permite elegir si está activada o no inicialmente
JCheckBox(String texto, **Icon** icono**)**	Crea una nueva casilla con el texto indicado y el icono que se elija
JCheckBox(String texto, **Icon** icono**, boolean** seleccionado**)**	Crea una nueva casilla con el texto indicado y el icono que se elija

imágenes

Se pueden crear distintas imágenes de una casilla de verificación al igual que ocurría con los botones. Los métodos de la clase **AbstractButton** (véase más arriba), permiten este hecho.

En el caso de las casillas de verificación, suele bastar con poner un icono inicial en el constructor o con el método **setIcon** y después asignar el icono que corresponde al estado de seleccionado de la casilla con **setSelectedIcon.**

eventos

Las casillas de verificación lanzan (al ser herederas de los botones), eventos **ActionEvent** cuando son seleccionadas. Pero disponen de un evento propio llamado **ItemEvent** que se lanza cuando se cambia el estado de una casilla (véase eventos *InputEvent*, página 140).

El interfaz relacionado es **ItemListener** y el método de captura es **itemStateChanged** que captura el evento cuando el estado de la casilla ha cambiado. El método **getItemSelectable** devuelve la casilla que produjo el evento, mientras que **getStateChanged** permite saber qué tipo de cambio ocurrió (ItemEvent.SELECTED o ItemEvent.DESELECTED)

Ejemplo:

```
public class VentanaCasillaAct extends JFrame implements
ItemListener {
 JCheckBox deportes, cultura;
 JLabel descripción;
 private String sdeportes="",scultura="";

 public VentanaCasillaAct(){
      Container conpane = getContentPane();
 conpane.setLayout(new FlowLayout());
      deportes = new JCheckBox("Deportes");
      cultura = new JCheckBox("Cultura");
      descripción = new JLabel("Tiene elegido: ");

      deportes.addItemListener(this);
      deportes.addItemListener(this);
      conpane.add(deportes);
      conpane.add(cultura);
      conpane.add(descripción);
}
public void itemStateChanged(ItemEvent e){
 if (e.getItemSelectable()==deportes){
      if (e.getStateChange()==ItemEvent.SELECTED)
           sdeportes=" deportes";
      else sdeportes="";
      }
      else { //sólo puede haberlo provocado el evento "Cultura"
           if (e.getStateChange()==ItemEvent.SELECTED)
                scultura=", cultura";
           else scultura="";
      }
      descripción.setText("Tiene elegido:"+sdeportes+scultura);
 }
}
```

botones de opción

Son casi iguales a los anteriores. Sólo que se utilizan para elegir una opción de entre un grupo de opciones. Como las casillas de verificación, la clase **JRadioButton** encargada de crear botones de radio, desciende de **JToggleButton.**constructores

constructor	uso
JRadioButton()	Construye un nuevo botón de radio

constructor	uso
JRadioButton(String texto**)**	Crea un nuevo botón de radio con el texto indicado
JRadioButton(Icon icon**)**	Crea un nuevo botón de radio con el icono indicado
JRadioButton(String texto, boolean activado**)**	Crea un nuevo botón de radio con el texto indicado y permite elegir si está activada o no inicialmente
JRadioButton(String texto, Icon icono**)**	Crea un nuevo botón de radio con el texto indicado y el icono que se elija
JRadioButton(String texto, Icon icono, boolean seleccionado**)**	Crea un nuevo botón de radio con el texto indicado y el icono que se elija

agrupar botones de radio

Para agrupar botones de radio y hacer que sólo se permita elegir uno de entre una lista de opciones, hay que utilizar la clase **ButtonGroup**.

Esta clase tiene un único constructor sin parámetros que crea un nuevo grupo de botones. Los botones así agrupados permitirán seleccionar sólo uno de la lista total

métodos

método	uso
void **add(**AbstractButton boton**)**	Añade el botón al grupo
void **remove(**AbstractButton boton**)**	Quita el botón del grupo

Con estos dos métodos se añaden botones de radio a un grupo y así sólo se podrá seleccionar una de las opciones.

eventos

Se manejan los mismos eventos que en las casillas de verificación. Ejemplo (igual que el de las casillas de activación):

```
public class VentanaBtRadio extends JFrame implements
ItemListener{
   JRadioButton deportes, cultura;
   ButtonGroup ocio=new ButtonGroup();
   JLabel descripción;

public VentanaBtRadio(){
   Container conpane = getContentPane();
   conpane.setLayout(new FlowLayout());
   deportes = new JRadioButton("Deportes");
   cultura = new JRadioButton("Cultura");
   descripción = new JLabel("Tiene elegido:");
```

```
   ocio.add(deportes);
   ocio.add(cultura);

   deportes.addItemListener(this);
   cultura.addItemListener(this);
   conpane.add(deportes);
   conpane.add(cultura);
   conpane.add(descripción);
}
public void itemStateChanged(ItemEvent e){
   if (e.getItemSelectable()==deportes){
         descripción.setText("Tiene elegido: deportes");
   }
   else { //sólo puede haber provocado el evento "Cultura"
         descripción.setText("Tiene elegido: cultura");
   }
}
}
```

viewport

Se trata de la clase madre de las clases que permiten desplazamientos (*scrolls*). Un *viewport* es una ventana dentro de la vista actual que muestra una sección de los datos y que permite desplazar la vista hacia el resto de datos.

La clase que representa los viewports es **JViewport**.

construcción

Un *viewport* se construye mediante un constructor por defecto. Una vez creado el objeto, necesitamos asignarle el componente ligero (un panel normalmente) sobre el que actuará el *viewport*. Esa asignación se realiza mediante el método **setView** al cual se le pasa el componente a visualizar mediante el viewport.

métodos interesantes

La clase **JViewport** posee una gran cantidad de métodos. Entre ellos destacan:

método	uso
void reshape(int x, **int** y, **int** ancho, **int** alto**)**	Asigna los límites del viewport
void setBorder(Border borde**)**	Asigna un borde al viewport
void setExtendSize(Dimension nueva**)**	Asigna el tamaño visible de la vista utilizando coordenadas de vista
void setView(Component panel**)**	Componente ligero sobre el que se aplica el viewport

método	uso
void **setViewPosition(**Point p**)**	Asigna las coordenadas de vista que aparecen en la esquina superior izquierda del viewport
void **setViewSize(**Dimension nueva**)**	Asigna como coordenadas de vista la esquina superior izquierda y el tamaño indicado
Dimension **toViewCoordinates(**Dimension tamaño**)**	Convierte tamaño en formato de coordenadas de puntos a tamaño en forma de coordenadas de vista
Point **toViewCoordinates(**Point tamaño**)**	Convierte punto en coordenadas de punto a coordenadas de vista

Todos los métodos **set** indicados en la tabla tienen versión **get** para obtener valores en lugar de asignar.

JScrollPane

Se trata de una clase espectacular que permite colocar barras de desplazamiento a cualquier componente. Usa, al igual que **Viewport**, la interfaz **Scrollable** que permite realizar desplazamientos.

constructores

constructor	uso
JScrollPane()	Construye un panel de desplazamiento vacío.
JScrollPane(Component c**)**	Construye un panel de desplazamiento para el componente **c**
JScrollPane(Component c, int políticaVertical, int políticaHorizontal**)**	Construye un panel para mostrar el componente c utilizando barras de desplazamiento. Las barras se configuran a través de los dos argumentos siguientes usando estas constantes estáticas: ◉ **VERTICAL_SCROLLBAR_ALLWAYS.** Barra vertical obligatoria ◉ **VERTICAL_SCROLLBAR_AS_NEEDED.** Saldrá la barra vertical cuando se necesite ◉ **VERTICAL_SCROLLBAR_NEVER.** Nunca saldrá la barra vertical ◉ **HORIZONTAL_SCROLLBAR_ALLWAYS** Barra horizontal obligatoria ◉ **HORIZONTAL_SCROLLBAR_AS_NEEDED** Saldrá la barra horizontal cuando se necesite ◉ **HORIZONTAL_SCROLLBAR_NEVER.** Nunca saldrá la barra horizontal.

métodos interesantes

método	uso
JScrollBar **getHorizontalScrollBar()**	Devuelve la barra de desplazamiento horizontal del panel
JScrollBar **getVerticalScrollBar()**	Devuelve la barra de desplazamiento vertical del panel.
Viewport **getViewport()**	Obtiene el *viewport* actual de la barra
void **setHorizontalScrollBar(** JScrollBar barraHorizontal**)**	Añade la barra que se pasa como argumento para que sea la barra horizontal del panel
void **setHVerticalScrollBar(** JScrollBar barraVertical**)**	Añade la barra que se pasa como argumento para que sea la barra vertical del panel
void **setVerticalScrollBarPolicy(**int políticaVertical**)**	Modifica el comportamiento de la barra vertical.
void **setHorizontalScrollBarPolicy (**int políticaHorizontal**)**	Modifica el comportamiento de la barra horizontal.

Barras de desplazamiento

La clase JScrollBar representa objetos de barra de desplazamiento. Normalmente es más que suficiente la clase anterior para controlar un componente. No obstante, es posible utilizar las barras de desplazamiento para acciones interesantes o para modificar las propiedades de las barras de un **JScrollPane.**

Las barras tienen estas propiedades:

- **valor.** Se trata del valor que actualmente representa la barra de desplazamiento.
- **extensión.** Es un valor que se refiere al tamaño de la guía (*track*) de la barra y al cambio de valores que se producirá en la misma si se hace clic entre la guía y los botones de la barra.
- **mínimo.** El mínimo valor que representa la barra. Es lo que *vale* la barra si la guía está al principio.
- **máximo.** El máximo valor que puede representar la barra. Es lo que *vale* la barra si la guía está al final.

construcción

constructor	uso
JScrollBar()	Construye una barra de desplazamiento vacía.
JScrollBar(int orientación**)**	Construye una barra de desplazamiento en la orientación indicada que puede ser: - JScrollBar**.HORIZONTAL** - JScrollBar**.VERTICAL**

constructor	uso
JScrollBar(int orientación, int valor, int extensión, int mínimo, int máximo**)**	Crea una barra de desplazamiento con la orientación, valor, extensión, valor

métodos interesantes

método	uso
void **setMaximum(**int máximo**)**	Ajusta el valor máximo de la barra
void **setMinimum(**int mínimo**)**	Ajusta el valor mínimo de la barra
void **setOrientation(**int orientación**)**	Cambiar la orientación de la barra
void **setValue(**int valor**)**	Ajusta el valor de la barra
void **setValues(**int valor, int extensión, int mínimo, int máximo**)**	Asigna las cuatro propiedades de la barra
void **setVisibleAmount(**int extensión**)**	Asigna la propiedad extensión del modelo

Todos esos métodos poseen la versión **get** para obtener valores.

eventos

Las barras generan eventos del tipo **AdjustmentEvent** cuando se modifican los valores de las barras. El método que controla estos eventos es **AdjustmentEvent**.

Por su parte el evento **AdjustmentEvent** posee dos métodos muy interesantes: **getValue()** que devuelve el valor de la barra y **getAdjustmentType()** que devuelve un entero que indica el tipo de cambio que se produjo en la barra. Este puede ser:

- AdjustmentEvent. **UNIT_INCREMENT.** Se pulsó en el botón de subir.
- AdjustmentEvent. **UNIT_DECREMENT** Se pulsó en el botón de bajar.
- AdjustmentEvent. **BLOCK_INCREMENT.** Se pulsó entre la guía y el botón de subir.
- AdjustmentEvent. **BLOCK_DECREMENT** Se pulsó entre la guía y el botón de bajar.
- AdjustmentEvent. **TRACK.** Se cambió la posición de la guía.

Ejemplo:

```
public class VentanaScrollBar extends JFrame implements
AdjustmentListener {
   JLabel etiqueta;
   JScrollBar barra;
   //constructor
public VentanaScrollBar() {
   barra=new JScrollBar(JScrollBar.HORIZONTAL,0, 30,0, 300);
   etiqueta=new JLabel("Valor: 0");
  etiqueta.setHorizontalAlignment(JLabel.CENTER);
  getContentPane().add(barra,BorderLayout.NORTH);
```

```
  getContentPane().add(etiqueta,BorderLayout.SOUTH);
  barra.addAdjustmentListener(this);
}

public void adjustmentValueChanged(AdjustmentEvent e) {
        etiqueta.setText("Valor: "+e.getValue());
    }
}
```

El resultado es:

deslizadores

La clase **JSlider** representa un tipo de objeto similar a las barras de desplazamiento pero pensado únicamente para elegir un valor numérico (al modo del ejemplo expuesto en las barras de desplazamiento).

construcción

constructor	uso
JSlider()	Crea un deslizador
JSlider(int orientación**)**	Crea un deslizador en la orientación indicada (JSlider.**HORIZONTAL** o JSlider.**VERTICAL**)
JSlider(int orientación, int mínimo, int máximo, int valor**)**	Crea el deslizador en la orientación señalada y con el mínimo, máximo y valor inicial señalados.

métodos

método	uso
void **setMaximum(**int máximo**)**	Ajusta el valor máximo de la barra
void **setMinimum(**int mínimo**)**	Ajusta el valor mínimo de la barra
void **setOrientation(**int orientación**)**	Cambiar la orientación de la barra
void **setValue(**int valor**)**	Ajusta el valor de la barra
void **setExtent(**int valor**)**	Cambia la extensión. La extensión es el rango de valores máximos a los que el deslizador no puede llegar. Si el valor máximo es 100 y la extensión es 40, el deslizador no podrá pasar de 60.

método	uso
void setInverted(boolean b**)**	Con valor **true** hace que los valores vayan desde el más alto al más bajo (al revés de lo habitual).
void setPaintLabels(boolean b**)**	Indica si se mostrarán las etiquetas del deslizador.
void setLabeltable(Dictionary etiquetas**)**	Permite especificar las etiquetas que se mostrarán en el deslizador.
void setPaintTicks(boolean b**)**	Indica si se mostrarán las marcas del deslizador.
void setPaintTrack(boolean b**)**	Indica si se pinta la guía del deslizador
void setSnapToTicks(boolean b**)**	Hace que la guía se ajuste automáticamente a las marcas.
void setMajorTickSpacing(int n**)**	Modifica el espaciamiento entre las marcas mayores del deslizador
void setMinorTickSpacing(int n**)**	Modifica el espaciamiento entre las marcas menores del deslizador

Hay métodos **get** que permiten obtener algunos de los valores ajustados (**getValue(), getOrientation(), getMajorTickSpacing(), getLabelTable(), getPaintTicks(),** etc.).

marcas y rellenos

Los deslizadores permiten mostrar marcas para facilitar al usuario la selección del valor requerido. Eso lo hace el método **setPaintTicks**, mientras que otros métodos permiten especificar el espacio entre las marcas y otras propiedades.

A su vez se puede utilizar esta sintaxis:

```
slider.putClientProperty("JSlider.isFilled", Boolean.TRUE);
```

Esto permite cambiar la propiedad cliente **isFilled** de los objetos JSlider y hacer así que se muestre un relleno

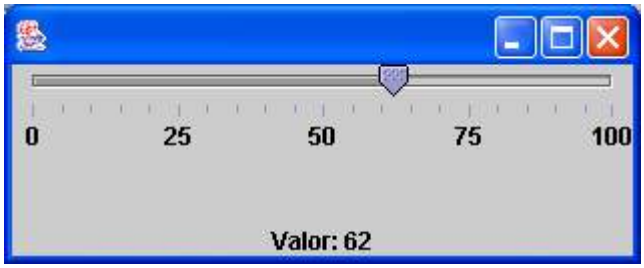
eventos

JSlider puede provocar eventos **ChangeEvent**, que deben ser capturados en clases que implementen la interfaz **ChangeListener**. Para que un deslizador lance eventos, debe usar el método **addChangeListener.** El método de captura de estos eventos es **stateChanged** que se producirá cuando se cambie de posición al deslizador. Ejemplo:

```
public class VentanaSlider extends JFrame implements
ChangeListener{
   JLabel etiqueta;
   JSlider barra;

public VentanaSlider() {
   barra=new JSlider(JSlider.HORIZONTAL,0,100,0);
```

```
    etiqueta=new JLabel("Valor: 0");
    etiqueta.setHorizontalAlignment(JLabel.CENTER);
    barra.addChangeListener(this);
    //apariencia del slider
    barra.putClientProperty("JSlider.isFilled", Boolean.TRUE);
    barra.setPaintLabels(true);
    barra.setPaintTicks(true);
    barra.setMajorTickSpacing(25);
    barra.setMinorTickSpacing(5);

    getContentPane().add(barra,BorderLayout.NORTH);
    getContentPane().add(etiqueta,BorderLayout.SOUTH);
}

public void stateChanged(ChangeEvent e) {
    etiqueta.setText("Valor: " +
          "((JSlider)e.getSource()).getValue());
}
```

El resultado es la siguiente ventana



listas

Son controles que permiten elegir entre un conjunto de alternativas. Al principio de muestra sólo un pequeño grupo de opciones. Se puede elegir una sola opción o varias si se hace clic manteniendo pulsada la tecla Control.

La clase de creación de listas es **JList**, heredera de **JComponent.**

construcción

constructor	uso
JList()	Crea un objeto de lista
JList(Object[] listData**)**	Crea una lista con el contenido del array de objetos, normalmente un array de **Strings**

Ejemplo:

```
JList animales=new JList(new String[]{"perro","gato","vaca",
  "oveja","cerdo","pollo","cabra","caballo","asno"});
```

métodos

método	uso
void clearSelection(int máximo**)**	Borra la selección actual
void ensureIndexIsVisible(int índice**)**	Desplaza el *viewport* de la lista para asegurar que se muestra el elemento número *índice*
int getFirstVisibleIndex()	Devuelve el primer número de índice visible en la lista
int getLastVisibleIndex()	Devuelve el último número de índice visible en la lista
int getMaxSelectionIndex()	Devuelve el índice máximo de la selección actual
int getMinSelectionIndex()	Devuelve el índice mínimo de la selección actual
Dimension getPreferredScrollableViewportSize()	Calcula el tamaño del *viewport* necesario para mostrar visualizar *visibleRowCount* filas
int getSelectionMode()	Indica si se puede seleccionar de forma múltiple o simple
***int* getSelectedIndex()**	Obtiene el número del primer índice seleccionado
***int*[] getSelectedIndices()**	Devuelve un array de números de índice de cada valor seleccionado.
***Object* getSelectedValue()**	Obtiene el primer valor seleccionado
***Object*[] getSelectedValues()**	Devuelve un array de valores para las celdas seleccionadas
boolean isSelectionEmpty()	**true** si no hay nada seleccionado
boolean is SelectedIndex(int i**)**	**true** si el índice señalado está seleccionado
void setFixedCellHeight(int alto**)**	Define la altura de cada celda de la lista
void setFixedCellWidth(int ancho**)**	Define la anchura de cada celda de la lista
void setSelectedIndex(int i**)**	Selecciona sólo la celda número *i*
void setSelectedIndices(int[] indices**)**	Selecciona los índices señalados
void setSelectionBackground(Color c**)**	Asigna color de fondo a las celdas seleccionadas
void setSelectionForeground(Color c**)**	Asigna color de texto a las celdas seleccionadas

método	uso
void setSelectionMode(int modo**)**	Permite cambiar el modo de selección de la lista. Puede tener como valores, constantes de la clase **ListSelectionModel**: ◉ **SINGLE_SELECTION.** Selección de una única opción ◉ **SINGLE_INTERVAL_SELECTION** Selecciona varias celdas, pero sólo si están seguidas ◉ **MULTIPLE_INTERVAL_SELECTION** Permite seleccionar cualquier número de celdas en cualquier orden
void setVisibleRowCount(int n**)**	Indica el número preferido de componentes que se pueden mostrar en la lista

asignación de barras de desplazamiento

A las listas se les pueden asignar paneles de desplazamiento como a cualquier otro componente, sólo que este es un componente especialmente propicio para hacerlo.

```
JScrollPane sp=new JScrollPane(animales);
```

eventos

Las listas se controlan con eventos **ListSelectionEvent,** en el paquete **javax.swing.event,** que se lanzan en listas que hagan uso del método **addListSelectionListener**. Las clases que deseen controlar esos eventos deben implementar la interfaz **ListSelectionListener** el cual obliga a definir el método **valueChanged** que será llamado cuando se modifique el valor de una lista.

Ejemplo:

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

public class VentanaLista extends JFrame implements
ListSelectionListener{
  JList lista=new JList(new String[]{"perro","gato","vaca",
        "oveja","cerdo","pollo","cabra","caballo","asno"});
  JLabel etiqueta=new JLabel("Seleccionado: ");
  JPanel p1=new JPanel(), p2=new JPanel();

  public VentanaLista() {
        JScrollPane sp=new JScrollPane(lista)
        p1.setLayout(new FlowLayout());
        p2.setLayout(new FlowLayout());
```

```
        lista.setVisibleRowCount(5);
        lista.addListSelectionListener(this);

        p1.add(sp);
        p2.add(etiqueta);
        getContentPane().add(p1,BorderLayout.CENTER);
        getContentPane().add(p2,BorderLayout.SOUTH);
    }
    public void valueChanged(ListSelectionEvent e) {
        String texto="Seleccionado: ";
        int i;
        //Se añade el texto del siguiente elemento
        //seleccionado hasta el último
        for(i=0;i<=lista.getSelectedValues().length-1;i++){
          texto+=(String) lista.getSelectedValues()[i]+" ";
        }
        etiqueta.setText(texto);
}
```

cuadros combinados

Son listas especiales que combinan las capacidades de una lista y un cuadro de texto. En apariencia son cuadros de texto. Pero un botón con una flecha permite abrirles para seleccionar una (y sólo una) opción. Es uno de los controles más populares de Windows.

constructores

constructor	uso
JComboBox()	Crea un objeto de lista
JComboBox(Object[] o**)**	Crea el cuadro combinado con el contenido del array de objetos, normalmente un array de **Strings**

métodos

método	uso
void **addItem(**Object o**)**	Añade el objeto *o* al final de la lista
Object **getItemAt(**int index**)**	Obtiene el elemento número *index* de la lista
int **getItemCount()**	Devuelve el número de elementos de la lista
Object **getSelectedItem()**	Obtiene el elemento seleccionado actual
void **hidePopup()**	Esconde la ventana emergente del cuadro

método	uso
void **insertItemAt(**Object objeto, int posición**)**	Añade el objeto en la posición indicada
boolean **isEditable()**	**true** si el cuadro de texto es editable
boolean **isPopupVisible()**	**true** si se puede ver la ventana emergente
void **removeAllItems()**	Borra todos los elementos de la lista combinada
void **removeItemAt(int** posición**)**	Quita el elemento situado en la posición indicada
boolean **selectWithKeyChar(char** c**)**	Selecciona el primer elemento de la lista que comienza con la letra indicada
void **setEditable(boolean** b**)**	Permite (o no) que el cuadro de texto sea editable
void **setMaximumRowCount(int** n**)**	Ajusta el número máximo de filas visibles. Si hay más filas, aparecerá una barra de desplazamiento.
void **setPopupVisible(boolean** b**)**	Despliega el cuadro de la lista
void **setSelectedIndex(int** i**)**	Selecciona el elemento de la lista número *i*
void **setSelectedItem(*Object* o)**	Selecciona el objeto marcado

eventos

Al ser un control mixto, que puede editar texto o seleccionar un elemento de la lista, envía dos tipos de eventos: **ActionEvent** (comentado anteriormente, en especial en el apartado de los botones) al pulsar Intro, e **ItemEvent** cuando se cambia un elemento de la lista (visto anteriormente en los controles de casilla de verificación).

El método de evento **getStateChange()** permite saber si el cambio fue para seleccionar (ItemEvent.SELECTED) o para deseleccionar (ItemEvent.DESELECTED). No suele ser necesario usarles ya que la clase **JComboBox** tiene elementos de sobra para trabajar.

Ejemplo(equivalente al ejemplo de la lista).

```
public class VentanaListaCombinada extends JFrame implements
ItemListener{
   JComboBox lista=new JComboBox(new String[]{"perro",
         "gato","vaca","oveja","cerdo","pollo","cabra",
         "caballo","asno"});
   JLabel etiqueta=new JLabel("Seleccionado: ");
   JPanel p1=new JPanel(), p2=new JPanel();

   public VentanaListaCombinada() {
         p1.setLayout(new FlowLayout());
         p2.setLayout(new FlowLayout());
```

```
        lista.setMaximumRowCount(5);
        lista.addItemListener(this);
        p1.add(lista);
        p2.add(etiqueta);
        getContentPane().add(p1,BorderLayout.CENTER);
        getContentPane().add(p2,BorderLayout.SOUTH);
    }
    public void itemStateChanged(ItemEvent e) {
        etiqueta.setText("Seleccionado:"+
              (String)lista.getSelectedItem());
    }
}
```

cuadros de diálogo Swing

cuadros de diálogo genéricos

La clase **JDialog** de Java permite crear cuadros de diálogo de propósito genérico. Esta es una de las clases más importantes de Swing. Permite crearse asignando como contenedor padre una ventana JFrame o un JApplet por ejemplo. Es más versátil pero más difícil de manejar. Sus métodos vienen heredados de JWindow. Entre ellos, el método **show** muestra el cuadro y **dispose** le cierra.

Además de la clase genérica, existen clases de diálogo planteadas para un solo propósito.

selectores de color

La clase **JColorChoose** permite obtener cuadros de selección de colores.

constructores

constructores	uso
JColorChooser()	Muestra un cuadro genérico de selección de colores
JColorChooser(Color c**)**	Muestra un cuadro genérico de selección de colores usando como color preseleccionado, el que define el parámetro.
JColorChooser(ColorSelectionModel csm**)**	Muestra un cuadro genérico de selección de colores usando el modelo de selección de colores indicado.

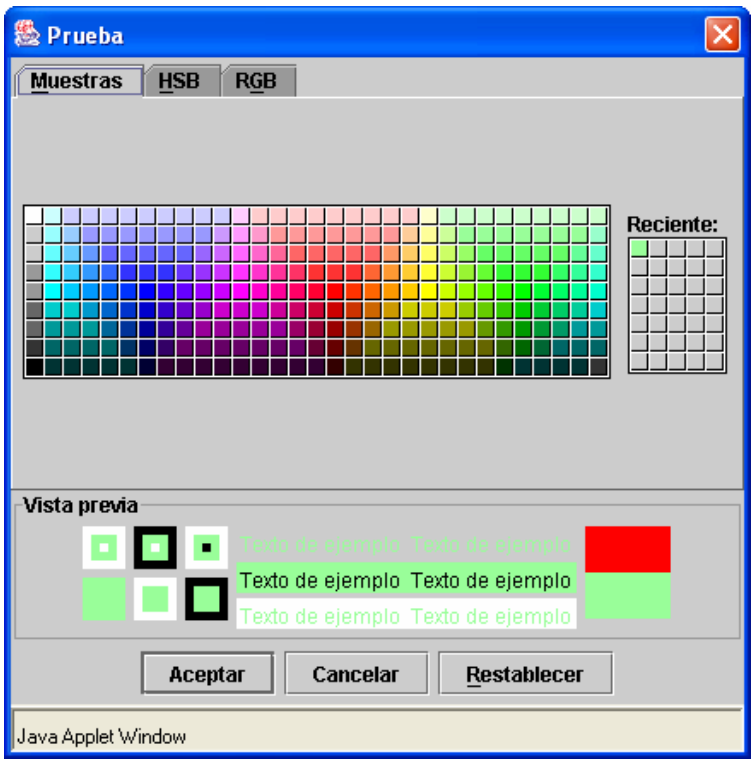


Ilustración 28, Cuadro de selección de colores

métodos

métodos	uso
AbstractColorChooserPanel[] **getChooserPanels()**	Obtiene un array con todos los paneles de selección de colores del cuadro.
static JDialog **createDialog(**Component padre, String título, boolean modal, JColorChooser jcc**,** ActionListenerk botónOK, ActionListener botónCancelar	Crea un cuadro de diálogo en el componente padre especificado con botones de Ok y Cancelar. Tendrá especificado por parámetros: el título, una indicación de si se desea que sea modal o no, el *JColorChooser* que irá incluido en el cuadro y dos indicaciones sobre qué objetos escucharán al botón Ok y al botón Cancelar.
Color **getColor()**	Obtiene el color actualmente seleccionado en el cuadro
JComponent **getPreviewPanel()**	Obtiene el panel de previsualización del color elegido.
ColorSelectionModel **getSelectionModel()**	Obtiene el modelo de selección de colores actual
AbstractColorChooserPanel **removeChooserPanel(** AbstractColorChooserPanel panel**)**	Elimina el panel especificado del cuadro

métodos	uso
void setChooserPanel(AbstractColorChooserPanel[] paneles**)**	Establece los paneles de selección a través del array que se pasa como parámetro.
void setColor(Color color**)**	Establece el color indicado como color actual del cuadro.
void setPreviewPanel(JComponent panel**)**	Establece el panel indicado como de previsualización de colores del cuadro.
void setSelectionModel(ColorSelectionModel csm**)**	Establece el modo de selección de colores del cuadro.
static Color showDialog(Component padre, String título, Color colorInicial**)**	Muestra un cuadro de selección de colores para el componente padre, con el título indicado y un primer color seleccionado. Si el usuario acepta el cuadro, devolverá el color elegido y si no devuelve **null.**

La forma sencilla de obtener un color mediante este cuadro es:

```
Color c=JColorChooser.showDialog(this,"Cambiar fondo",
        Color.RED);
```

Selección de archivos

La clase **JFileChooser** se utiliza para seleccionar archivos. Su funcionamiento es muy parecido al cuadro anterior

constructores

constructores	uso
JFileChooser()	Crea un nuevo cuadro de selección de archivos
JFileChooser(String ruta**)**	Utiliza una cadena de texto como ruta predeterminada para el cuadro de selección de archivos
JFileChooser(String ruta, FileSystemView vista**)**	Igual que el anterior pero utiliza además el objeto de vista de archivos indicada
JFileChooser(File ruta**)**	Utiliza la ruta dada en forma de objeto File como ruta predeterminada para el cuadro de selección de archivos
JFileChooser(File ruta, FileSystemView vista**)**	Igual que el anterior pero utiliza además el objeto de vista de archivos indicada
JFileChooser(FileSystemView vista**)**	Crea un selector de archivos con el objeto de vista de archivos indicado

métodos

métodos	uso
void addActionListener (ActionListener al**)**	Aplica un objeto *oyente* de eventos de acción al cuadro

métodos	uso
void addChosableFileFilter(FileFilter ff)	Añade un filtro de archivos al cuadro de selección de archivos
void approveSelection()	Es llamado automáticamente cuando el usuario acepta el cuadro pulsando "Abrir" o "Guardar"
void cancelSelection()	Es llamado automáticamente cuando el usuario cancela el cuadro
void ChangeToParentDirectory()	Hace que el cuadro cambie al directorio padre del que mostraba hasta ese momento
void ensureFileIsVisible(File f)	Asegura que el archivo *f* sea visible en el cuadro
FileFilter getAcceptAllFileFilter()	Obtiene el filtro de sistema referido a todos los archivos de la carpeta (en Windows es la expresión *.*)
FileFilter[] getChoosableFileFilters()	Obtiene la lista de todos los filtros que el usuario puede escoger en el cuadro
File getCurrentDirectory()	Obtiene el directorio actual en forma de objeto *File*
String getDescription(File f)	Obtiene la cadena que describe el tipo de archivo al que pertenece *f*
int getDialogType()	Indica el tipo de cuadro que es el selector de archivo, puede ser: ◉ **JFileChooser.SAVE_DIALOG** ◉ **JFileChooser.OPEN_DIALOG** ◉ **JFileChooser.CUSTOM_DIALOG**
FileFilter getFileFilter()	Obtiene el filtro de archivos que se aplica actualmente al cuadro
FileSystemView getFileSystemView()	Obtiene el objeto de vista de archivos de sistema actual
FileView getFileView()	Obtiene el objeto de vista de archivos actual
Icon getIcon(*File* f)	Devuelve el icono del archivo
Icon getName(*File* f)	Obtiene el nombre del archivo
File getSelectedFile()	Obtiene el archivo seleccionado
File[] getSelectedFiles()	Devuelve la lista de archivos seleccionados
String getTypeDescription(File f)	Obtiene una cadena descriptiva del tipo de archivos al que pertenece f
boolean isAcceptAllFileFilter()	**true** si el filtro general (*.* en Windows) está seleccionado
boolean isDirectorySelectionEnabled()	**true** si se permiten seleccionar carpetas
boolean isFileHiddingEnabled()	**true** si no se muestran los archivos ocultos en el cuadro
boolean isFileSelectionEnabled()	Indica si se permite seleccionar archivos

métodos	uso
boolean isMultiSelectionEnabled()	Indica si se permite la selección múltiple de elementos en el cuadro
boolean isTraversable(File f)	**true** si se puede entrar en el directorio representado por el objeto *f*
void rescanCurrentDirectory()	Refresca el contenido de la carpeta actual
void resetChoosableFileFilter()	Restaura el filtro de archivos a su posición inicial
void setApproveButtonMnemonic(char c)	Activa el carácter *c* para que sea la tecla de acceso rápido al botón de aceptar
void setApproveButtonText(String texto)	Establece el texto del botón de aprobación del cuadro
void setControlButtonsAreShown(boolean b)	Establece si se muestran los botones de aprobación y cancelación en el cuadro
void setCurrentDirectory(*File* f)	Hace que la carpeta representada por el objeto *f* se considere la carpeta actual del cuadro
void setDialogTitle(String título)	Establece el título del cuadro de diálogo
void setDialogType(int tipo)	Establece el tipo de cuadro de diálogo. Las posibilidades son: ◉ **JFileChooser**.SAVE_DIALOG **Guardar** ◉ **JFileChooser**.OPEN_DIALOG **Abrir** ◉ **JFileChooser**.CUSTOM_DIALOG **Personal**
void setFileFilter(FileFilter ff)	Establece el filtro de archivos actual
void setFileHiddingEnabled(boolean b)	Indica si se muestran los archivos ocultos
void setFileSelectionEnabled(boolean b)	Indica si se permite selección de archivos en el cuadro
void setFileSelectionMode(int modo)	Indica el modo de selección de archivos del cuadro. Puede ser: ◉ **JFileChooser.FILES_ONLY** Sólo archivos ◉ **JFileChooser.DIRECTORIES_ONLY.** Sólo directorios ◉ **JFileChooser.FILES_AND_DIRECTORIES.** Ambas cosas
void setFileSystemView(FileSystemView fsv)	Establece el tipo de vista de archivos de sistema. Lo cual indica si se ven unidades de disco, las carpetas de sistema ,...

métodos	uso
void **setFileView(**FileView fv**)**	Establece el tipo de vista de. Lo cual indica qué tipo de iconos se ven por ejemplo
void **setMultiSelectionEnabled(**boolean b**)**	Establece la selección múltiple de archivos (si se indica un **true** como argumento)
void **setSelectedFile(**File f**)**	Establece *f* como el archivo o carpeta actualmente seleccionado en el cuadro
void **setSelectedFiles(**File[] lista**)**	Hace que la lista de archivos se muestre como conjunto de archivos seleccionados en el cuadro
int **showDialog(**Component c, String texto**)**	Crea un cuadro de diálogo asociado al selector de archivos preparado con un botón de aprobación que posee el texto indicado.
int **showOpenDialog(**Component padre**)**	Crea un cuadro de apertura de archivo
int **showSaveDialog(**Component padre**)**	Crea un cuadro de guardado de archivo

Ejemplo:

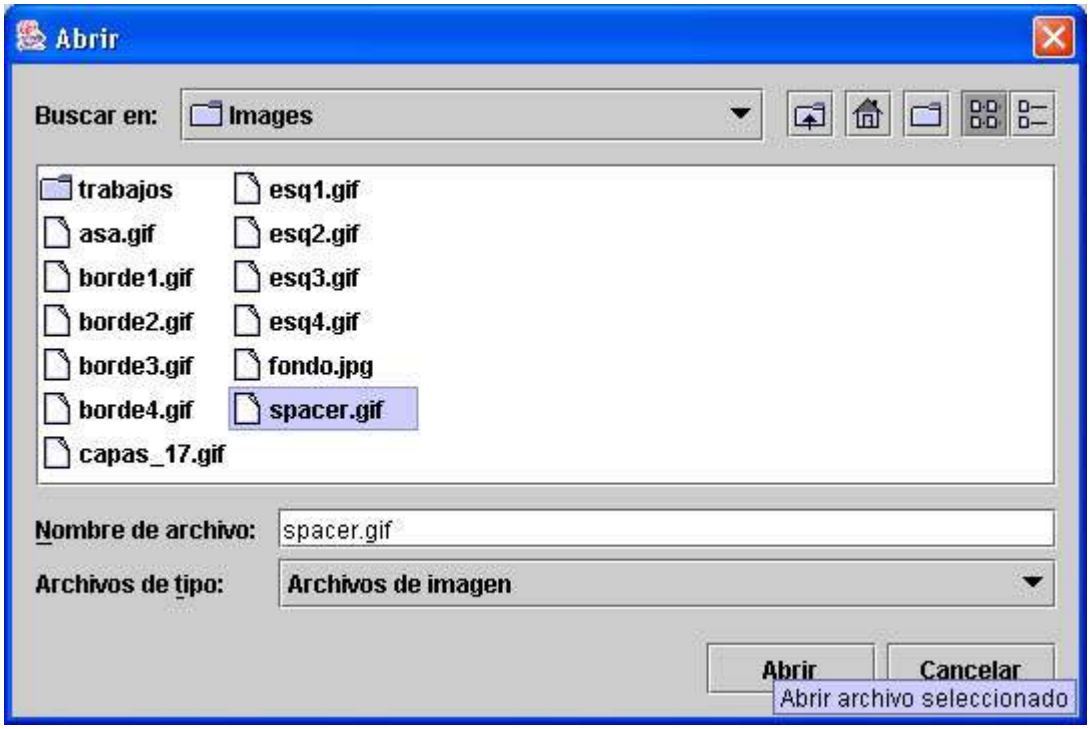
```
JFileChooser selector = new JFileChooser();
ImgFileFilter filtro= new ImgFileFilter();
selector.setFileFilter(filtro);
int returnVal = selector.showOpenDialog(null);
if(returnVal == JFileChooser.APPROVE_OPTION) {
System.out.println("Se eligió el archivo " +
                      selector.getSelectedFile().getName());
```

Para que este ejemplo funcione, hay que definir la clase *ImgFileFilter*. Gracias a esa clase se consigue (como se verá en el siguiente ejemplo), que se muestren sólo archivos JPG y GIF.

Eso es lo que se llama un filtro. Los filtros se crean usando clases derivadas de la clase abstracta **javax.swing.filechooser.FileFilter**. Las clases de filtros deben incluir un método **accept** que devuelva verdadero para todos los archivos que se deben mostrar en el cuadro y un método **getDescription** que devuelva una cadena indicando el tipo de archivos que el filtro es capaz de mostrar. La definición sería:

```
private class ImgFileFilter extends FileFilter{
        public boolean accept(File f) {
            if (f.getPath().endsWith(".jpg") ||
                f.getPath().endsWith(".gif")||
                f.getPath().endsWith(".jpeg")||
                f.isDirectory()) return true;
            else return false;
        }
        public String getDescription() {
            return "Archivos de imagen"; }  }
```

El cuadro sería:



El archivo elegido, no se abre. Se captura con **getSelectedFile** y es el programa el que tiene que abrirlo o tratarlo desde el código.

applets

introducción

Sin duda uno de los pilares de Java es su dedicación a la programación en red. Esta programación crea aplicaciones distribuida desde una red. La lentitud de Internet ha propiciado que las aplicaciones distribuidas sigan siendo problemáticas. Por ello se tiende a que la aplicación que se distribuye sea sencilla para conseguir que llegue lo antes posible.

De hecho la idea es que un usuario no perciba diferencia alguna entre una página con Java y una página sin Java. Esta idea no está conseguida del todo, pero sí se ha realizado un importante avanza mediante esos subprogramas Java incrustados dentro del código normal (HTML) de una página web. Estos subprogramas son llamados **applets**.

Un applet es un programa Java que tiene ciertas particularidades derivadas de la idea de tener que colocar ese programa en una página web. Para que la página web HTML muestre el subprograma Java, tiene que incluir una etiqueta **applet** que es la encargada de asociar el archivo **class** que representa al subprograma con la página. Esa misma etiqueta determina la posición y el tamaño del applet

```
<applet code="applet1.class" width="250" height="300">
</applet>
```

Probar la applet implicar abrir la página, aunque el JDK de Java incluye un programa llamado **appletviewer** para mostrar el applet.

En definitiva los pasos para crear un applet son:

1> Crear una página web dejando hueco para colocar el applet de Java.

2> Crear el código de la applet y compilarlo en su archivo **class** correspondiente.

3> Integrar la applet en la página insertando una etiqueta **applet** en la posición en la que se desea la applet. Los atributos **width** y **height** permiten especificar el tamaño de la misma.

4> Al publicar la página se deben enviar los archivos HTML junto con los archivos **class** y los archivos necesarios para que la página y applet se vean correctamente (imágenes, hojas de estilo,...).

probar applets. *Appletviewer*

Para probar applets basta con abrir la página web en la que se creó el applet. No obstante como esto a veces es muy pesado, el SDK de Java incorpora un visor llamado AppletViewer.

Para trabajar con este visor basta compilar el archivo java para obtener el precompilado **class**. Después se llama al programa **appletviewer** pasando como parámetro el archivo java:

```
c:\ejemplos>appletviewer applet.java
```

El resultado es una ventana que muestra el applet sin usar página web.

En el código java se puede incluir la etiqueta **applet** de HTML a fin de probar el applet con un tamaño y configuración concretos. Para ello se coloca la etiqueta en los comentarios del archivo:

```
import javax.swing.JApplet;
/* <applet code="applet.class"
        width=200 height=200>
  </applet>
*/
public class applet extends JApplet{
...
```

navegadores sin posibilidad de applets

Los navegadores que no dispongan de capacidad de interpretar applets, desconocerán las etiquetas applet y no ejecutarán su contenido. Para estos navegadores se coloca un texto que avisa de la situación al usuario. Este texto es ignorado por los navegadores que sí tengan capacidad de interpretar código applet.

Ejemplo:

```
<applet code="applet3.class" width=400 height=250>
  Su navegador no tiene capacidad de mostrar applets.
</applet>
```

clases Applet y JApplet

En AWT hay una clase pensada para crear applets. La clase se llama Applet y deriva de la clase **Panel** que, a su vez, deriva de **Container.** Swing creo una subclase a partir de ésta para asegurar la compatibilidad con los componentes Swing. Esta clase es JApplet.

Lógicamente si se usa la clase **Applet** para crear los subprogramas, se deben usar componentes AWT en su interior; si se usa **JApplet** se deben usar sólo componentes Swing.

compatibilidad

Un problema inherente a la naturaleza de la web, es que no podemos saber el software que posee el usuario. Si la red utilizada para acceder a la página con applet es Internet, entonces no tendremos ninguna seguridad de qué versión de intérprete de Java posee el usuario. De hecho ni siquiera sabremos si puede ejecutar subprogramas Java.

Esto provoca los siguientes problemas:

- La mayoría de los navegadores utilizan de forma nativa la versión 1.1 de Java. Esta versión no incluye Swing. La solución para ellos sería crear Applets AWT, pero se desperdician las nuevas capacidades de Java.
- Se pueden deshabilitar las características java de los navegadores. Aquellas personas que realicen esta acción, no podrán visualizar ninguna aplicación Java.

- Aún avisando a nuestros usuarios de que deben poseer una versión más moderna de Java, éstos no se animarán a descargar un entorno de ejecución más moderno, bien porque consideran que el problema es ajeno a ellos, bien porque las descargas siguen siendo muy lentas desde Internet, o bien por que tienen miedo de que sea un *truco* de un pirata que intenta introducirnos software dañino.

Sun proporciona dos soluciones para instalar plataformas de ejecución Java (JRE, *Java Runtime Environment*) y evitar estos problemas en lo posible.

el *Java Plugin*, el conector de Java

Convierte las etiquetas **<applet>** en etiquetas **<embed>** u **<object>**.que permiten hacer pensar a Explorer que el contenido de la página es un control ActiveX y a Netscape Navigator y compatibles que se trata de un plug-in externo. Esto se conseguía con una aplicación llamada **HTMLConverter.**

Desde la versión 1.4 no es necesario el HTMLConverter. El plugin funciona perfectamente en la versión 6 y superior de Netscape y en Explorer 4 o superior. El programa que conseguía esto era el **HTMLConverter.** Desde la versión 1.3.1_01a del plugin se dejó de utilizar el convertidor, ya que ahora funciona la conversión aún utilizando la etiqueta **applet.**

La descarga de este software ocupa unas 5 MB (de ahí la reticencia de los usuarios a su instalación). En entornos de Intranet es cómodo su uso ya que el programador lo instala en las máquina y el problema se remedia fácilmente. El plugin está disponible en las direcciones:

- **http://www.java.com/es/download** (español)
- **http://www.java.com/en/download** (inglés).

De todas las formas cuando un usuario llega a una página con applets creadas con versiones superiores al plugin de Java instalado, entonces se le pedirá descargar el plugin. Otra cuestión es que el usuario permita esa descarga.

Java Web Start

Su tamaño es casi idéntico al del Java plug-in. La diferencia está en que ejecuta las applets desde fuera del navegador, en una aplicación independiente. Para descargarlo basta ir a **http://java.sun.com/products/javawebstart** (también se incluyen en el kit de desarrollo de Java J2SE).

Una ventaja que posee respecto al *plugin* es el hecho de que la seguridad es más avanzada (se permite manipular ficheros aunque de forma controlada, mediante el constante permiso del usuario).

Las aplicaciones creadas para **Web Start** están creadas en un protocolo llamado **JNPL**. Este protocolo está basado en XML.

métodos de una applet

Un applet tiene un ciclo de vida que está directamente relacionado con cuatro métodos definidos en la clase **applet** que es la clase padre de **JApplet**. Además posee métodos comunes con las clases Applet y Panel (al ser heredera de ellas)

método *init*

Este método es llamado automática tras crearse el applet. Aquí se prepara el programa, los recursos necesarios, los elementos GUI, etc. **No se deben realizar estas operaciones en el constructor.** El constructor de un **applet** no está pensado para esto, sólo está para crear en sí el applet.

método *start*

Es llamado cuando el programa se hace visible. Se le llama tantas veces sea necesario.

método *stop*

Es llamado cuando el programa se hace invisible. Es decir, cuando se cierra la ventana de la página web en la que está incrustado el applet o cuando el usuario acude a otra página.

Es este método se debería detener las actividades innecesarias. Son innecesarias aquellas que no debería seguir realizándose cuando el programa está detenido; de otro modo se consumen recursos inútiles en el ordenador del usuario.

Por otro lado hay veces en las que puede interesar que el programa continúe realizando operaciones en segundo plano.

método *destroy*

Es el método que se implementa para eliminar todos los objetos antes de que el applet desaparezca. Cerrar sockets, eliminar objetos GUI, ... son tareas habituales de este método. El método es llamado cuando se elimina del todo; esto es difícil de establecer ya que cada navegador realiza esta tarea en diferentes momentos.

otros métodos

métodos	uso
Container getContentPane()	Obtiene el *content pane* de la applet. Es ahí donde se suelen colocar el resto de los elementos de las applets
Component getGlassPane()	Devuelve el *glass pane*
JMenuBar getJMenuBar()	Devuelve la barra de menús
JLayeredPane getLayeredPane()	Devuelve el *layered pane* de la applet
JRootPane getRootPane()	Obtiene el *root pane*
void remove(Component c)	Elimina el componente *c* de la applet
void setContentPane(Container c)	Hace que el contenedor *c* sea el actual *content pane* de la applet
void setGlassPane(Component c)	Hace que el componente *c* sea asignado como *glass pane* de la applet
void setJMenuBar(JMenuBar menu)	Asigna un menú a la ventana
void setLayeredPane(JLayeredPane l)	Cambia el *layered pane* de la applet

métodos	uso
void **setLayout(**LayoutManager l**)**	Cambia la disposición de la página por la indicada por l (véase administración de diseño, página 193)
void **repaint()**	Llama al método **paint**

métodos heredados de la clase Applet

AppleContext **getAppleContext()**	Obtiene el objeto *Apple Context* relacionado con la applet
String **getAppletInfo()**	Obtiene información sobre la applet. Esta información incluye autor, versión etc.
URL **getCodeBase()**	Obtiene la URL base de la applet
URL **getDocumentBase()**	Obtiene la URL de la página en la que se coloca la applet
Locale **getLocale()**	Obtiene los valores locales de la applet (si se configuró)
String **getParameter(**String nombre**)**	Obtiene el parámetro llamado *nombre* pasado por la etiqueta **applet** de la página web
String[][] **getParameterInfo()**	Obtiene un array bidimensional de información sobre los parámetros pasados por la applet. Este método se suele sobrescribir para indicar correctamente la información que debe retornar.
boolean **isActive()**	**true** si la applet está activa
void **resize(**Dimension d**)**	Pide cambiar el tamaño de la applet
void **resize(**int x, int y**)**	Pide cambiar el tamaño de la applet
void **setStub(**AppleStub stub**)**	Estable el *stub* de la applet
void **showStatus(**String mensaje**)**	Escribe en la barra de estado

la etiqueta *applet*

La etiqueta **applet** tiene diversos atributos para controlar adecuadamente la carga y ejecución del applet. Los atributos (se usan así:

```
<applet atributo1='valor' atributo2='valor'...>
```

Los posibles atributos a utilizar son:

atributo	significado
ARCHIVE	Indica el archivo JAR en el que está colocado el archivo class indicado en el atributo code

atributo	significado
CODEBASE	URL que indica el directorio en el que se busca el código de la *applet*. Si la applet está en un paquete o en un archivo JAR, este atributo hace referencia al directorio que contiene a ese paquete o archivo JAR.
CODE	Ruta a la applet desde el directorio anterior, o desde el directorio en el que está la página (si no se indicó *codebase*). Debe incluir la extensión **class**
ALT	Texto alternativa a mostrar por los navegadores que no han cargado el applet por alguna razón
NAME	Nombre del la applet en el navegador. Esto permite que las applets interactúen entre sí
WIDTH	Anchura de la applet en la página
HEIGHT	Altura de la applet en la página
ALIGN	Alineación de la applet respecto a los elementos que la siguen. Los valores que más se usan para este atributo son **left** y **right**
VSPACE	Espacio se deja en vertical alrededor de la applet
HSPACE	Espacio se deja en horizontal alrededor de la applet

parámetros

Desde la etiqueta **applet** se pueden pasar parámetros a una applet. Eso facilita reutilizar applets de una aplicación a otra, ya que se puede personalizar el resultado de la misma gracias a estos parámetros.

De esto se encarga una etiqueta HTML que debe ir en el interior de las etiquetas **applet.** La etiqueta en cuestión se llama **param** y tiene dos atributos: **name** que indica el nombre del parámetro y **value** que posee el valor inicial de ese parámetro.

Desde el programa Java, es la función **getParameter** procedente de la clase **Applet** la que captura el valor de un parámetro. Devuelve siempre un String, por lo que se deben convertir los valores dentro del código Java si queremos usar número o fechas por ejemplo. Ejemplo:

Código en la página web:

```
<applet code=" applet1.class"
 width=350 height=100>
    <param name='texto' value='Esto es una prueba'>
 </applet>
```

Código Java

```
public class applet2 extends JApplet {
    public void init() {
        JLabel l=new JLabel(getParameter("texto"));
        getContentPane().add(l);
    }
}
```

El resultado será una página que muestra el texto *Esto es una prueba*.

manejar el navegador desde la applet

Desde la propia applet se puede acceder a recursos del propio navegador. Esto permite una interesante comunicación.

línea de estado

La línea de estado del navegador es la franja gris inferior de la ventana en la que el navegador dispone diversa información. La applet puede requerir escribir en esa barra haciendo uso del método **showStatus,** al cual se le pasa el texto que se desea escribir.

cambiar de documento en el navegador

Mediante el método **getAppletContext(),** se obtiene una variable de tipo **AppleContext**. *AppleContext* es un interfaz relacionado con las applets que proporciona diversos métodos (la mayoría están integrados en las clases Applet).

Quizá el más interesante sea **showDocument**. Este método recibe un objeto URL (se hablará de ellos más adelante) que contiene una ruta. Esa ruta URL es enviada al navegador que hace que desde ese instante se convierta en la página visible. Ejemplo:

```
try{
     URL u=new URL("http://www.jorgesanchez.net");
     getAppletContext().showDocument(u);

}
catch(MalformedURLException mfe){
   JOptionPane.showConfirmDialog(this,"Error al cambiar ruta");
  }
```

La creación de archivos URL se debe capturar recogiendo la excepción **MalformedURLException** perteneciente al paquete **java.net** (al igual que la clase URL).

Una versión del método **showDocument** permite usar un segundo parámetro de tipo String para indicar en qué marco se muestra el documento. Las posibilidades son las mismas que posee el atributo **target** de la etiqueta **a** del lenguaje HTML: **_self** (en el propio marco) **, _top** (ocupando todo el área de la página) **, _parent** (ocupando el marco padre) **, _blank** (usando una nueva ventana vacía) o un nombre que se corresponde al identificador de un marco de la página web.

comunicación entre applets

Los applets de la misma página se pueden comunicar usando variables **AppletContext**, ya que poseen un método llamado **getApplet** que recibe un String en el que se indica el nombre del applet y devuelve un objeto de tipo **Applet** que apunta al objeto representado por ese applet:

```
Applet ap2=getAppletContext().getApplet("applet2");
```

El nombre utilizado debe corresponderse con el atributo **name** de la etiqueta **applet** en la página web.

paquetes

Las clases se agrupan en paquetes como ya se ha comentado anteriormente en este manual. Eso se suele respetar también cuando el applet es publicado en la red. Pero hay que tener en cuenta que se tiene que respetar esa estructura.

Se utiliza el atributo **codebase** de la etiqueta **applet** para indicar desde donde comienza la ruta del archivo class. El inicio debe ser la raíz de las applets. A partir de ahí se indica el nombre completo (incluyendo el nombre del paquete) de clase en el atributo **code**. Ejemplo:

```
<applet codebase="http://www.jorgesanchez.net/"
  code="utiles.relojes.analogico.class">
```

También es válido:

```
<applet codebase="http://www.jorgesanchez.net/"
  code="utiles/relojes/analogico.class">
```

archivos JAR

El SDK de Java posee una utilidad muy poderosa que permite comprimir varias clases en un archivo que posee extensión JAR. Normalmente se empaqueta toda una aplicación en cada archivo JAR.

En el caso de las applets, se recomienda mucho su uso ya que reducen notablemente la carga al ser archivos comprimidos. Además facilitan el mantenimiento de las applets ya que hay que publicar un solo archivo.

Su uso es sencillo, hay que entender que un archivo JAR es un tipo especial de paquete. El compilador del SDK entiende su uso, las applets no tienen problema tampoco en usarlo e incluso se pueden colocar otros tipos de archivos (como imágenes, audio,...) dentro del conjunto comprimido JAR y se pueden recuperar utilizando **getClass().getResource(**rutaArchivo**)**

El programa **jar** del SDK es el que realiza esta operación. Pero casi todos los entornos de programación poseen facilidades de creación de archivos JAR. Ejemplos:

```
jar cvf archivo.jar archivo1.class archivo2.class
  Comprime las dos clases en un solo archivo
jar tvf archivo.jar
  Lista el contenido del archivo jar
jar xvf archivo.jar
  descomprime el archivo. Extrae su contenido
```

manifesto

Los archivos JAR incluso pueden incluir carpetas dentro e incluir un texto descriptivo llamado manifiesto (**manifesto**). Gracias a este texto se le pueden dar funcionalidades avanzadas a los archivos JAR. El manifiesto es un archivo de texto que contiene líneas

que van colocadas en pares Nombre-Valor. Se encuentra dentro de la carpeta **META-INF** dentro de los archivos JAR.

Entre las indicaciones que se pueden utilizar están: el algoritmo de cifrado, nombre de clases, creador del manifesto, clases importadas,...

Ejemplo:

```
Manifest-Version: 1.0
Created-By: NetBeans IDE
Specified-By: pruebaJAR/a1.jarContent

Name:H41.class
RevisionNumber:4.0

Name:H42.class
RevisionNumber:2.0
```

Más información en **http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html**

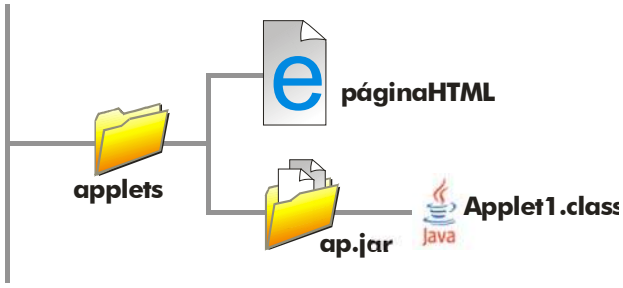
uso de archivos JAR en las páginas web

Se realiza utilizando el atributo **archive** de la etiqueta **applet:**

```
<applet code="reloj" archive="utilidades.jar">
```

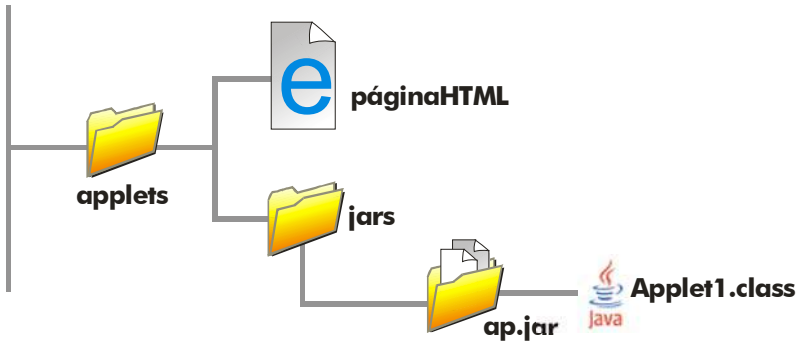
Ejemplos:

1> Si tenemos en el paquete **applets** la página web y dentro un archivo **JAR** que contiene el archivo class, **Applet1.class**



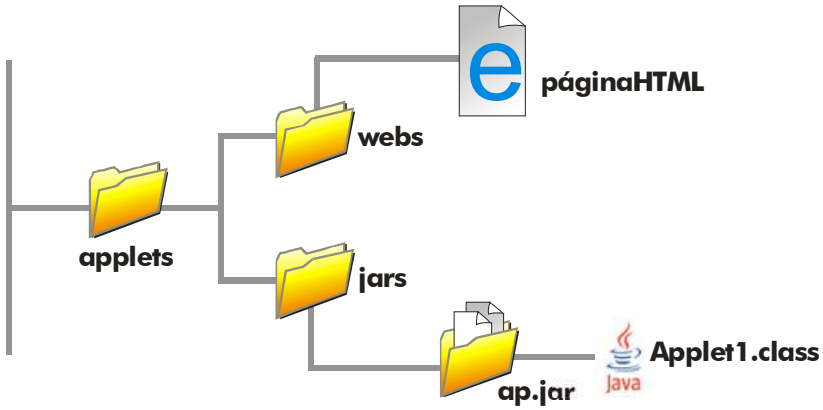

```
<applet code=applets.Applet1.class archive=ap.jar>
```

2> La misma situación, pero ahora hay una carpeta llamada **jars** en la que se encuentra el archivo **jar:**




```
<applet code=applets.Applet1.class archive=jars/ap.jar>
```

3> Ahora la página web se encuentra en una carpeta y el archivo **jar** en otra que parten de la misma raíz




```
<applet code=applets.Applet1.class archive=../jars/ap.jar>
```

el administrador de seguridad

Se trata del programa conocido como *Security Manager*. Está incluido dentro de los navegadores y sirve para controlar las operaciones realizadas por las applets. Los usuarios pueden configurarlo con lo que es imposible saber la configuración de seguridad en cada ordenador.

En un caso normal este administrador prohíbe:

- Leer y escribir archivos en el ordenador del usuario
- Abrir conexiones de red hacia un ordenador distinto del que se utilizó para crear el applet
- No pueden ejecutar programas en el ordenador del usuario.

applets firmados

Se trata de una posibilidad interesante. Permite autentificar al autor del applet y de esta forma conseguir que haya applets de confianza a los que se les permite un acceso menos restringido al sistema.

Esta autentificación se consigue por medio de una firma digital. Esta firma autentifica al creador de la applet y así, si el usuario confía en ese creador, se podrán realizar operaciones que, de otro modo, prohibiría el administrador de seguridad.

Las firmas se basan en un sistema de codificación de datos por clave pública, lo cual significa que la firma en realidad es una clave pública que sirva para codificar nuestros datos. Una segunda clave (privada) permitirá decodificarlos. El usuario receptor de la applet recibe esa clave pública.

La cuestión es ¿cómo podemos estar seguros de que esa clave pertenece a quien dice pertenecer? Es decir, cómo se verifica la autenticidad de la clave. Eso se realiza mediante certificados de autenticidad (CA). Estos certificados son emitidos por una entidad emisora que verifica que esa clave pertenece realmente a quien dice pertenecer. Esto significa que la confianza se traspasa ahora a esta entidad.

Podría surgir la duda de si esa entidad es válida. El problema puede continuar intentando verificar la identidad del CA, pero ahora las posibilidades son cada vez más reducidas al haber menos claves a verificar.

certificados de sitio

Estos certificados se envían al navegador para validar las firmas de los archivos JAR. Estos certificados se pueden guardar para asignarles más o menos privilegios cuando utilicemos sus applets.

certificados de usuario

Estos se envían en dirección contraria, es decir al sitio. Son los certificados que verifican la identidad del usuario.

cómo se consiguen certificados

Uno mismo genera sus claves pública y privada. Después se envía la pública a una entidad verificadora de certificados que creará (previo envío de información y, a veces, de pago) su certificado.

firmar archivos

keytool es una utilidad que viene con el SDK de Java que permite manejar toda una base de datos de identidades. Mientras que **jarsigner** permite firmar los archivos **jar** que se deseen. Se puede obtener información de:

- http://java.sun.com/products/jdk/1.2/docs/tooldocs/win32/keytool.html
- http://java.sun.com/products/jdk/1.2/docs/tooldocs/win32/jarsigner.html

almacenes de claves. Comando keytool

keytool es un programa gratuito que permite exportar certificados. Para ello tenemos que poseer el certificado en un archivo CER. Después hay que almacenarle en un almacén de certificados. Normalmente se almacenan los certificados en el archivo .keystore que está en la carpeta de usuario. Pero se puede cambiar. Ejemplo:

```
c:\>keytool -import -file jorge.cer -alias jorge -storepass
mimono
```

Esto importa el certificado y lo almacena en el almacén por defecto con el alias *jorge* y la contraseña *mimono*. Después nos pregunta la utilidad **keytool** si podemos confiar en el certificado, al decir que sí, estamos diciendo que confiamos en el certificado.

Sin embargo este certificado instalado sólo vale para confiar en él. Si deseamos incorporar nuestro certificado para firmar, entonces necesitamos conseguir un certificado que incluya la clave privada (*keytool* tiene capacidad para conseguir claves privadas). Con nuestra clave pública y privada deberemos acudir a un organismo emisor y así tendremos un archivo que valida nuestras contraseñas. Para añadirle:

```
c:\>keytool -import -alias Jorge -file Jorge.x509 -keypass
palencia -storepass mimono
```

palencia es la contraseña que nos permite acceder a la clave privada, *mimono* es la contraseña para almacenar.

Más tarde podemos firmar un archivo JAR de esta forma:

```
C:\>jarsigner -storepass mimono archivo.jar Jorge
```

En el archivo META-INF del archivo JAR se añadirá una entrada Jorge.SF (firma del archivo) y Jorge.DSA (firma binaria real).

Para probar se puede crear una autofirma (aunque sin validez al no haber certificador), de esta forma:

```
c:\> keytool -genkey -alias Jorge2 -file Jorge2.csr -keypass
palencia -storepass mimono
```

programación en red

introducción

Sin duda la red es el contexto de trabajo fundamental de java. Lo mejor de Java está creado para la red. El paquete **java.net** es el encargado de almacenar clases que permitan generar aplicaciones para redes. En él podremos encontrar clases orientadas a la programación de sockets y herramientas de trabajo con URLs.

También se utiliza mucho el paquete **java.io (**visto en el tema dedicado a la entrada y salida, página 93). Esto se debe a que la comunicación entre clientes y servidores se realiza intercambiando flujos de datos, por lo que las clases para controlar estos flujos son las mismas que las vistas en el tema citado.

sockets

Son la base de la programación en red. Se trata de el conjunto de una dirección de servidor y un número de puerto. Esto posibilita la comunicación entre un cliente y un servidor a través del puerto del socket. Para ello el servidor tiene que estar *escuchando* por ese puerto.

Para ello habrá al menos dos aplicaciones en ejecución: una en el servidor que es la que abre el socket a la escucha, y otra en el cliente que hace las peticiones en el socket. Normalmente la aplicación de servidor ejecuta varias instancias de sí misma para permitir la comunicación con varios clientes a la vez.

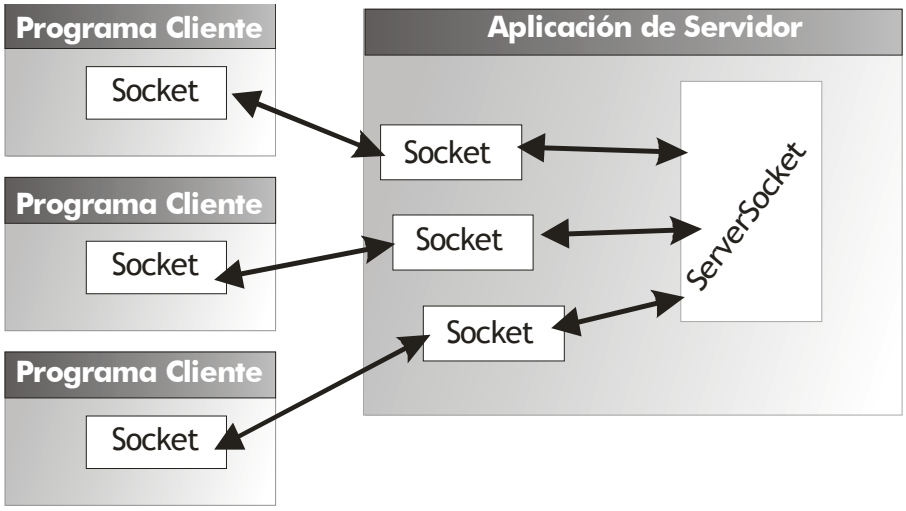


Ilustración 29, Esquema de la comunicación con sockets

clientes

Las aplicaciones clientes son las que se comunican con servidores mediante un socket. Se abre un puerto de comunicación en ordenador del cliente hacia un servidor cuya dirección ha de ser conocida.

La clase que permite esta comunicación es la clase **java.net.Socket**.

construcción de sockets

constructor	uso
Socket(String servidor, int puerto**)** throws IOException, UnknownHostException	Crea un nuevo socket hacia el servidor utilizando el puerto indicado
Socket(InetAdress servidor, int puerto**)** throws IOException	Como el anterior, sólo que el servidor se establece con un objeto **InetAddress**
Socket(InetAdress servidor, int puerto, InetAdress dirLocal, int puertoLocal**)** throws IOException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.
Socket(String servidor, int puerto, InetAdress dirLocal, int puertoLocal**)** throws IOException, UnknownHostException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.

Ejemplo:

```
try{
   Socket s=new Socket("time-a.mist.gov",13);
}
catch(UnknownHostException une){
   System.out.println("No se encuentra el servidor");
}
catch(IOException une){
   System.out.println("Error en la comunicación");
}
```

Es necesario capturar esas excepciones. La excepción **UnknownHostException** es una subclase de **IOException** (a veces se captura sólo esta última para simplificar el código ya que si sólo se incluye ésta, se captura también a la anterior; aunque suele ser más conveniente separar los tipos de error),

lectura y escritura por el socket

El hecho de establecer comunicación mediante un socket con un servidor, posibilita el envío y la recepción de datos. Esto se realiza con las clases de entrada y salida. La clase Socket proporciona estos métodos:

método	uso
InputStream **getInputStream()** throws IOException	Obtiene la corriente de entrada de datos para el socket
OutputStream **getOutputStream()** throws IOException	Obtiene la corriente de salida de datos para el socket

Se puede observar como se obtienen objetos de entrada y salida orientados al byte. Si la comunicación entre servidor y cliente se realiza mediante cadenas de texto (algo muy

habitual) entonces se suele convertir en objetos de tipo **BufferedReader** para la entrada (como se hacía con el teclado) y objetos de tipo **PrintWriter** (salida de datos en forma de texto):

```
try{
   Socket socket=new Socket("servidor.dementiras.com",7633);
   BufferedReader in=new BufferedReader(
         new InputStreamReader(socket.getInputStream()));
   PrintWriter out=new PrintWriter(
         socket.getOutputStream(),true)); // el parámetro
                     //true sirve para volcar la salida al
                     //dispositivo de salida (autoflush)
   boolean salir=false;
   do {
         s=in.readLine();
         if(s!=null) System.out.println(s);
         else salir=true;
   }while(!salir);
}
catch(UnknownHostException une){
   System.out.println("No se encuentra el servidor");
}
catch(IOException une){
   System.out.println("Error en la comunicación");
}
```

servidores

En el caso de un programa de servidor, éste se ha de ocupar de recibir el fuljo de datos que procede del socket del cliente (además tiene que procurar servir a varios clientes a la vez).

Para que un programa abra un socket de servidor. Se usa la clase **ServerSocket** cuyo constructor permite indicar el puerto que se abre:

```
ServerSocket socketServidor=new ServerSocket(8341);
```

Después se tiene que crear un socket para atender a los clientes. Para ello hay un método llamado **accept** que espera que el servidor atienda a los clientes. El funcionamiento es el siguiente, cuando el socket de servidorEste método obtiene un objeto Socket para comunicarse con el cliente. Ejemplo:

```
try{
   ServerSocket s=new ServerSocket(8189);
   Socket recepcion=s.accept();
   //El servidor espera hasta que llegue un cliente
```

```
  BufferedReader in=new BufferedReader(new
                        InputStreamReader(
                              recepcion.getInputStream()));
  PrintWriter out=new PrintWriter(
                        recepcion.getOutputStream(),true);
  out.println("Hola! Introduzca ADIOS para salir");
   boolean done=false;
  while(!done){
       String linea=in.readLine();
       if(linea==null) done=true;
       else{
             out.println("Echo: "+linea);
             if (linea.trim().equals("ADIOS")) done=true;
        }
   }
   recepcion.close();
}catch(IOException e){}
```

Este es un servidor que acepta texto de entrada y lo repite hasta que el usuario escribe ADIOS. Al final la conexión del cliente se cierra con el método **close** de la clase Socket.

servidor de múltiples clientes

Para poder escuchar a más de un cliente a la vez se utilizan threads lanzado un thread cada vez que llega un cliente y hay que manipularle. En el thread el método **run** contendrá las instrucciones de comunicación con el cliente.

En este tipo de servidores, el servidor se queda a la espera de clientes. cada vez que llega un cliente, le asigna un Thread para él. Por lo que se crea una clase derivada de la clase **Thread** que es la encargada de atender a los clientes.

Es decir, el servidor sería por ejemplo:

```
public static void main(String args[]){
  try{
       //Se crea el servidor de sockets
       ServerSocket servidor=new ServerSocket(8347);
       while(true){//bucle infinito
         //El servidor espera al cliente siguiente y le
        //asigna un nuevo socket
        Socket socket=servidor.accept();
        //se crea el Thread cliente y se le pasa el socket
        Cliente c=new Cliente(socket);
        c.start();//se lanza el Thread
       }
   }catch(IOException e){}
```

Por su parte el cliente tendría este código:

```
public class Cliente extends Thread{
    private Socket socket;
    public Cliente(Socket s) {
      socket=s;
    }

    public void run(){
        try{
             //Obtención de las corrientes de datos
            BufferedReader in=new BufferedReader(
                  new InputStreamReader(
                        socket.getInputStream()));
            PrintWriter out=new PrintWriter(
                        socket.getOutputStream(),true);

            out.println("Bienvenido");
            boolean salir=false;//controla la salida
            while (!salir){
               resp=in.readLine();//lectura
               ...//proceso de datos de lectura
               out.println("....");//datos de salida
               if(...) salir=true;//condición de salida
              }
              out.println("ADIOOOOOOS");
              socket.close();
        }catch(Exception e){}

    }
```

métodos de Socket

método	uso
void **setSoTimeout(**int tiempo**)**	Establece el tiempo máximo de bloqueo cuando se está esperando entrada de datos por parte del socket. Si se cumple el tiempo, se genera una interrupción del tipo: **InterruptedIOException.**
void **close()**	Cierra el socket
boolean **isClosed()**	**true** si el socket está cerrado

método	uso
void **shutdownOutput()**	Cierra el flujo de salida de datos para que el servidor (en aquellos que funcionan de esta forma) sepa que se terminó el envío de datos. Disponible desde la versión 1.3
void **shutdownInput()**	Cierra el flujo de entrada de datos. Si se intenta leer desde el socket, se leerá el fin de archivo. Desde la versión 1.3

clase InetAddress

Obtiene un objeto que representa una dirección de Internet. Para ello se puede emplear este código:

```
InetAddress dirección=InetAddress.getByName(
                    "time-a.nist.gov");
System.out.println(dirección.getHostAddress());//129.6.15.28
```

Otra forma de obtener la dirección es usar el método **getAddress** que devuelve un array de bytes. Cada byte representa un número de la dirección.

A veces un servidor tiene varios nombres. Para ello se usa:

```
InetAddress[] nombres =
        InetAddress.getAllByName("www.elpais.es");
System.out.println(nombres.length);
for(int i=0;i<nombres.length;i++) {
  System.out.println(nombres[i].getHostAddress());
}
//Escribe:
//195.176.255.171
//195.176.255.172
```

lista de métodos

método	uso
static InetAddress **getByName(**String servidor**)**	Obtiene el objeto *InetAddress* que corresponde al servidor indicado
static InetAddress **getAllByName(**String servidor**)**	Obtiene todos los objetos *InetAddress* asociados al servidor indicado
static InetAddress **getByAddress(**byte[] direcciónIP**)**	Obtiene el objeto *InetAddress* asociado a esa dirección IP
static InetAddress **getLocalHostName()**	Obtiene el objeto *InetAddress* que corresponde al servidor actual
String **getHostAddress()**	Obtiene la dirección IP en forma de cadena
byte[] **getAddress()**	Obtiene la dirección IP en forma de array de bytes
String **getHostName()**	Obtiene el nombre del servidor

método	uso
String **getCanonicalHostName()**	Obtiene el nombre canónigo completo (suele ser la dirección real del host

conexiones URL

Realizar conexiones mediante sockets tiene el inconveniente de que hay que conocer las conexiones a un nivel de funcionamiento bastante bajo. Por eso se utilizan también conexiones a nivel más alto mediante objetos URL

objetos URL

Un objeto URL representa una dirección alcanzable de una red TCP/IP. Su construcción se suele realizar de esta forma:

```
URL url=new URL("http://www.elpais.es");
```

El método **openStream** obtiene un objeto **InputStream** a través del cual se puede obtener el contenido.

Ejemplo:

```
try{
   URL url=new URL("http://www.elpais.es");
   BufferedReader in=new BufferedReader(
         new InputStreamReader(url.openStream()));
   String linea;
   while((linea=in.readLine())!=null)
   {
         System.out.println(linea);
   }
}
catch(MalformedURLException mue){
    System.out.println("URL no válida");
}
catch(IOException ioe){
    System.out.println("Error en la comunicación");
}
```

El ejemplo anterior lee la página de portada de El Pais. Como se ve se captura una excepción de tipo **MalformedURLException** esta excepción se requiere capturar al construir el nuevo objeto URL. Se trata de una excepción hija de **IOException** (que en el ejemplo también es capturada).

constructores

constructor	uso
URL(String url**)** throws MalformedURLException	Construye un objeto URL a partir de la ruta dada
URL(String protocolo, **String** servidor, **String** archivo**)** throws MalformedURLException	Construye un objeto URL con los parámetros desglosados que se observan
URL(String protocolo, **String** servidor, **int** puerto, **String** archivo**)** throws MalformedURLException	Construye un objeto URL con los parámetros desglosados que se observan

métodos

constructor	uso
int getDefaultPort()	Devuelve el puerto asociado por defecto para la URL del objeto
int getPort()	Devuelve el puerto que utiliza realmente el objeto URL
String getHost()	Devuelve el nombre del servidor
String getQuery()	Devuelve la cadena que se envía al archivo para ser procesado por el (es lo que sigue al signo ? de una dirección URL)
String getPath()	Obtiene una cadena con la ruta hacia el archivo desde el servidor y el nombre completo del archivo
String getFile()	Igual que la anterior, pero además añade lo que devuelve **getQuery.**
String getUserInfo()	Devuelve la parte con los datos del usuario de la dirección URL
URLConnection openConnection()	Obtiene un objeto de tipo **URLConnection** que permite establecer una conexión completa con el servidor (véase capítulo siguiente)
InputStream openStream()	Permite establecer una corriente de entrada para recibir el recurso
boolean sameFile(URL url2**)**	Compara la URL del argumento con la original y devuelve **true** si se refieren al mismo archivo
String toExternalForm()	Devuelve una cadena que representa al objeto URL

objetos URI

Hay una distinción entre URL y URI. Los URI definen recursos sintácticos de Internet. Esos recursos no tienen necesidad de poseer datos para localizar, esa es su diferencia. De hecho una URL es una caso de URI en el que los datos son localizables (es decir una

URL hace referencia a datos que existen, una URI es teórica, los datos podrían no existir).

Por esta razón la clase URL de Java sólo usa recursos FTP o HTTP. La clase URI permite examinar direcciones y desglosarlas en sus distintos apartados (servidor, puerto, etc.).

Los métodos de esta clase son muy similares a los de URL (la mayoría son los mismos). Además un método llamado **toURL** convierte el URI en URL.

JEditorPane

Se trata de una clase que permite mostrar páginas web de forma muy fácil. Es una clase heredera de **JTextComponent** y sirve para mostrar documentos HTML (hasta versión 3.2), RTF y de texto plano.

construcción

construcción	uso
JEditorPane()	Crea un nuevo panel de edición
JEditorPane(String url**)**	Crea un nuevo panel que muestra el documento contenido en la cadena (que debe implementar una URL)
JEditor(URL url**)**	Crea y muestra un panel con el contenido de la URL indicada
JEditor(String mime, String texto**)**	Establece el editor para incluir documentos del tipo MIME indicado y que contendrán el texto inicial marcado

métodos

construcción	uso
addHyperlinkListener(HyperlinkListener oyente**)**	Indica qué objeto escuchará los eventos de tipo **HiperlinkEvent** que esta función creará
String **getContentType()**	Devuelve una cadena con el tipo de contenido que el editor es capaz de mostrar.
HiperlinkListeners[] **getHyperlinkListeners()**	Obtiene una lista, en forma de array, de los objetos que actualmente actúan de oyentes de eventos *Hiperlink* del editor
URL **getPage()**	Devuelve el objeto URL que está mostrando ahora el editor
void **scrollsToReference(**String marca**)**	Mueve la pantalla hasta colocarse en la posición de la marca indicada (es el mismo tipo de elemento que el devuelto por el método **getReference** de la clase **URL**)

construcción	uso
void setContentType(String tipo**)**	Establece el tipo de contenido que el editor visualizará. Puede valer una de estas cadenas: ◉ **text/plain.** Permite mostrar texto plano (sin formato). ◉ **text/html.** Permite mostrar texto HTML 3.2 ◉ **text/rtf.** Permite mostrar documentos RTF
void setEditable(boolean siONo**)**	Indica si el contenido del panel podrá ser editado. Para mostrar texto HTML (y sólo mostrar) es necesario el valor **false.**
void setPage(URL url**)** throws IOException	Hace que el editor muestre la página contenida en la URL indicada
void setPage(String url**)** throws IOException	Igual que el anterior, sólo que la URL se pasa en forma de cadena (menos recomendable)
void setText(String texto**)**	Hace que el editor muestre el texto indicado

eventos *Hyperlink*

Son eventos asociados a esta clase (véase también ***eventos InputEvent***, página 143). Ocurren cuando el usaurio realiza cualquier acción sobre uno de los enlaces de la página que muestra el editor. Es el interfaz **HyperlinkListener** el que implementa el método **hyperlinkUpdate** encargado de manejar estos eventos.

Los eventos **HyperlinkEvent** son lanzados cuando el usuario realiza cualquier operación sobre ellos. Hay que utilizar el método **getEventType** de la clase **HyperlinkEvent** para saber que tipo de evento fue el producido. El objeto devuelto por *getEventType* es un objeto de tipo **HyperlinkEvent.EventType** que es de una subclase dentro de la *HyperlinkEvent*.

Para determinar el valor de este objeto, se ha de comparar con las constantes:

- **HyperlinkEvent.EventType.ENTERED.** Si el ratón está encima del enlace
- **HyperlinkEvent.EventType.EXITED.** Si el ratón sale fuera del enlace
- **HyperlinkEvent.EventType.ACTIVATED.** Si se hizo clic sobre el enlace

Por otro lado, el método **getURL** devuelve la URL del enlace usado. Este es el método más utilizado de este tipo de eventos. Por último, **getDescription** devuelve el texto del enlace.

Ejemplo de uso de eventos de enlace:

```
public void hyperlinkUpdate(HyperlinkEvent e){
 try{
       HyperlinkEvent.EventType tipo=e.getEventType();
       if (tipo== HyperlinkEvent.EventType.ENTERED)
```

```
            objetoEditorPane.setCursor(
               Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
         else if (tipo== HyperlinkEvent.EventType.EXITED)
           objetoEditorPane.setCursor(Cursor.getDefaultCursor());
         else objetoEditorPane.setPage(e.getURL());
 }
 catch(IOException e){}
}
```

conexiones URLConnection

Los objetos **URLConnection** permiten establecer comunicaciones más detalladas con los servidores. Los pasos son:

1> Obtener el objeto de conexión con el método **openConnection** de la clase URL

2> Establecer propiedades para comunicar:

método	uso
void **setDoInput(**boolean b**)**	Permite que el usuario reciba datos desde la URL si *b* es **true** (por defecto está establecido a *true*)
void **setDoOutput(**boolean b)	Permite que el usuario envíe datos si *b* es **true** (éste no está establecido al principio)
void **setIfModifiedSince(**long tiempo**)**	Sólo muestra recursos con fecha posterior a la dada (la fecha se da en milisegundos a partir de 1970, el método **getTime** de la clase **Date** consigue este dato).
void **setUseCaches(**boolean b**)**	Permite recuperar datos desde un caché
void **setAllowUserInteraction(**boolean b**)**	Permite solicitar contraseña al usuario. Esto lo debe realizar un programa externo, lo cuál no tiene efecto fuera de un applet (en un navegador, el navegador se encarga de sacar el cuadro).
void **RequestProperty(**String clave, String valor**)**	Establece un campo de cabecera

3> Conectar (método **connect**)

método	uso
void **connect()**	Conecta con el recurso remoto y recupera información de la cabecera de respuesta

4> Solicitar información de cabecera

método	uso
String **getHeaderFieldKey(**int n**)**	Obtiene el campo clave número *n* de la cabecera de respuesta
String **getHeaderField(**int n**)**	Obtiene el valor de la clave número *n*
int **getContentLength()**	Recupera el tamaño del contenido
String **getContentType**	Recupera una cadena que indica el tipo de contenido
long **getDate()**	Fecha del recurso
long **getExpiration()**	Obtiene la fecha de expiración del recurso
long **getLastModifier()**	Fecha de última modificación

5> Obtener la información del recurso

método	uso
InputStream **openInputStream()**	Obtiene un flujo para recibir datos desde el servidor (es igual que el método **openStream** de la clase URL)
OutputStream **openOutputStream()**	Abre un canal de salida hacia el servidor

Gracias a esto las conexiones son más poderosas y se permiten muchas más operaciones.

Ejemplo:

```
try{
   URL url=new URL("http://www.elpais.es");
   URLConnection conexión=url.openConnection();
   conexión.setDoOutput(true);
   conexión.connect();

   //Lectura y muestra de los encabezados
   int i=1;
   while(conexión.getHeaderFieldKey(i)!=null){
         System.out.println(conexión.getHeaderFieldKey(i)+
                           ":"+conexión.getHeaderField(i));
         i++;
   }

   //Lectura y muestra del contenido
   BufferedReader in=new BufferedReader(
         new InputStreamReader(conexión.getInputStream()));
   String s=in.readLine();
```

```
  while(s!=null){
        System.out.println(s);
        s=in.readLine();
   }
}
catch(Exception e){
   System.out.println("Fallo");
}
```

JDBC

introducción

SGBD

Una de las principales aplicaciones de cualquier lenguaje moderno es la posibilidad de utilizar datos pertenecientes a un sistema de base de datos. La dificultad del manejo de archivos y las facilidades de manejo de datos que ofrecen los sistemas gestores de base de datos (SGBDs) son los causantes de esta necesidad.

En el mercado hay gran cantidad de bases de datos y cada una de ellas se maneja de un modo diferente. Esto está en contra del planteamiento fundamental de Java que intenta que la programación sea independiente de la plataforma.

Hoy en día hay que tener en cuenta que la inmensa mayoría de los SGBD administran **bases de datos relacionales**. Éstas son bases de datos que permiten organizar los datos en tablas que después se relacionan mediante campos clave y que trabajan con el lenguaje estándar conocido como SQL.

Cada tabla es una serie de filas y columnas, en la que cada fila es un **registro** y cada columna un **campo.** Cada campo representa un dato de los elementos almacenados en la tabla (nombre, DNI,...). Cada registro representa un elemento de la tabla (la señora Eva Jiménez , el señor Andrés Gutiérrez,...). No puede aparecer dos veces el mismo registro, por lo que uno o más campos forman lo que se conoce como **clave principal.** La clave principal no se puede repetir en dos registros y permite que los datos se relacionen.

En cualquier caso en este manual no se pretende revisar cómo funcionan las bases de datos relacionales, sólo se explica cómo acceder desde Java a este tipo de bases de datos.

La idea de Sun era desarrollar una sola API (*application programming* interfaces, interfaz de programación de aplicaciones) para el acceso a bases de datos, esta interfaz se conoce como JDBC (*java data base connect)*. Los requisitos eran:

- JDBC sería una API a nivel SQL (el lenguaje SQL sería el que realizaría la conexión con las bases de datos), independiente por tanto de la plataforma
- JDBC debía ser similar al funcionamiento de las API para acceso a bases de datos existentes (en especial a la ya entonces famosa ODBC de Microsoft)
- JDBC debía ser sencilla de manejar

ODBC

Es imposible en este apartado no dedicar un pequeño espacio a ODBC *(open data base connectivity)*. Se trata del interfaz diseñado por Microsoft como estándar para el manejo de datos de diversas bases de datos.

Esta API de bases de datos se ha convertido en la más popular. Su éxito se basa en la facilidad de instalación y configuración en Windows y en que casi todos los gestores de bases de datos la utilizan. Pero tiene varios inconvenientes:

- No es transportable a todos los sistemas.

- Está creada en C, con los problemas que supone eso para la programación en otros lenguajes (punteros **void** por ejemplo). Esto, en definitiva, supone que si no se utiliza C se han de diseñar librerías intermedias de acceso a ODBC; con lo que se multiplican las capas dificultando la programación.

- Es compleja su programación.

No obstante, debido a la popularidad de ODBC, existen puentes ODBC para JDBC que permiten comunicar bases de datos con controladores ODBC con aplicaciones programadas para JDBC.

estructura JDBC

En el diagrama siguiente se puede apreciar como la idea es que las aplicaciones sólo se tengan que comunicar con el interfaz JDBC. Éste es el encargada de comunicarse con los sistemas de base de datos.

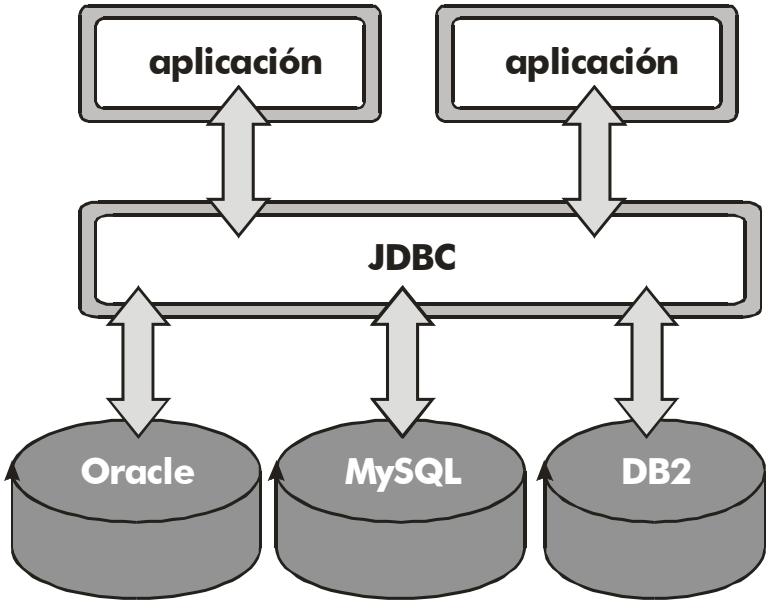


Ilustración 30, Esquema JDBC

Esto hace que la programación de aplicaciones de acceso a bases de datos no necesite conocer el funcionamiento del SGBD en particular, lo que hay que conocer es el funcionamiento de JDBC. Por supuesto, es necesario adquirir un controlador JDBC para el sistema gestor de base de datos que utilicemos. La comunicación fundamental entre las aplicaciones y JDBC se realiza a través de instrucciones SQL.

controladores

Una vez instalado, configurado y puesto en funcionamiento nuestro sistema gestor de base de datos favorito, si queremos que las bases de datos creadas por él sean accesibles desde los programas Java, necesitamos el controlador JDBC de ese sistema.

Hay cuatro tipos de controladores:

- **Tipo 1**. Controlador que traduce de JDBC a ODBC, Un controlador de este tipo es la pasarela JDBC-ODBC. No es muy productiva ya que necesita ser configurada

para un controlador ODBC concreto. aunque actualmente existen mejores soluciones de este tipo, sigue requiriendo tener dos controladores en el cliente.

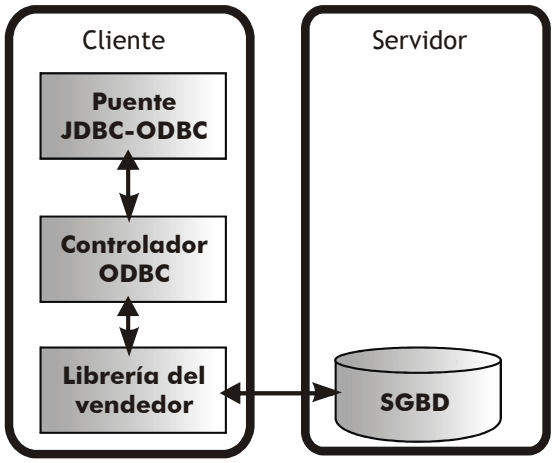


Ilustración 31, Funcionamiento del controlador JDBC de tipo 1

- **Tipo 2**. Son controladores parcialmente escritos en Java y parcialmente escritos en el código nativo que comunica con el API de la base de datos. En estos controladores hay que instalar tanto paquetes Java como paquetes de código nativo, lo que no les hace válidos para Internet.

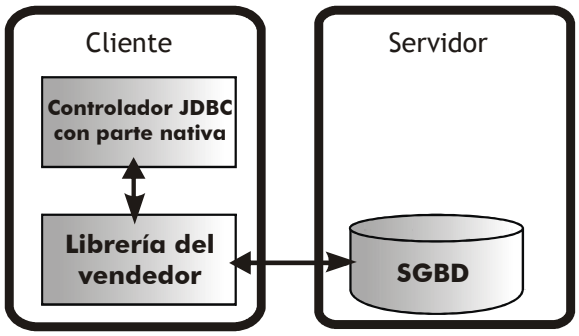


Ilustración 32, Funcionamiento del controlador JDBC de tipo 2

- **Tipo 3**. Son paquetes puros de Java que usan protocolos independientes de la base de datos. Es decir las instrucciones JDBC son pasadas a un servidor genérico de base de datos que utiliza un protocolo determinado. No requiere tener instalado ningún software de librería de la base de datos. Lo malo es que el servidor intermedio (**middleware**) suele requerir instrucciones específicas de la base de datos.

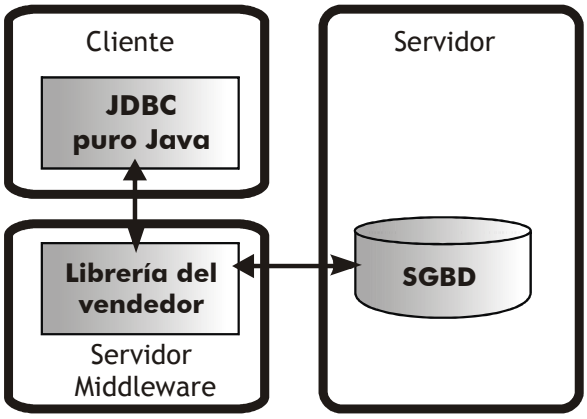


Ilustración 33, Funcionamiento del controlador JDBC de tipo 3

- **Tipo 4**. Paquetes de Java puro que traducen peticiones JDBC a protocolo de base de datos específico. No requieren intermediarios entre el software JDBC y la base de datos

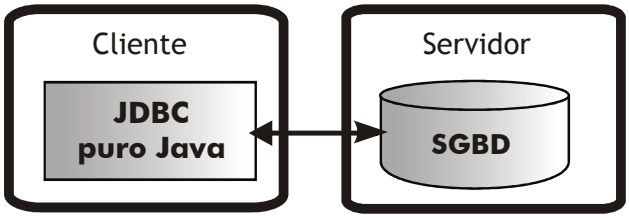


Ilustración 34, Funcionamiento del controlador JDBC de tipo 4

Normalmente las distribuciones JDBC que suministran los fabricantes son de tipo 3 o 4. para adquirir estos controladores es necesario ponerse en contacto con el fabricante o dirigirse a su página web y después descargarlo. Las instrucciones de instalación las da el fabricante, pero en caso de ser controladores de tipo 3 o 4 habrá que instalar los paquetes del API JDBC en la ruta Classpath para que sean accesibles por los compiladores Java.

Para saber si existe controlador JDBC para nuestra base de datos de trabajo, se puede comprobar en la dirección:

http://servlet.java.sun.com/products/jdbc/drivers/index.html

conexión

Para conseguir conectar una base de datos con una aplicación, nuestra aplicación requiere el URL de la base de datos y las propiedades que establezca nuestro controlador JDBC. Las clases necesarias para usar JDBC están en el paquete **java.sql.**

El primer paso es instalar el controlador (*driver*) de la base de datos. Hay varias posibilidades una es colocar el controlador en el atributo *jdbcdrivers* de la máquina virtual al ejecutar el programa:

```
java -Djdbc.drivers=com.mysql.jdbc.Driver
```

Pero el método que más se usa es lanzar el controlador en la propia aplicación mediante el método estático **forName** de la clase **Class**. Para evitar problemas con la máquina virtual se utiliza **newInstance**. Es decir, el formato es:

```
Class.forName(rutaDelDriver).newInstance();
```

La ruta del driver tiene que venir en las instrucciones del fabricante. Por ejemplo en el caso del controlador MySQL el formato es:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Esa instrucción puede dar lugar a las excepciones **ClassNotFoundException** (si no se encontró la clase en el drivr JDBC), InstantiationException (si no se puede crear el driver para la base de datos) e **IllegalAccessException** (si el acceso a la base de datos no fue correcto). Se suelen capturar todas ellas con una **Exception** genérica.

Una vez que el controlador se ha registrado, entonces se abre la URL a la base de datos. cuyo formato suele ser:

```
jdbc:sgbd://servidor/basedatos:puerto
```

Por ejemplo en MySQL

```
jdbc:sgbd://localhost/prueba:3306
```

La conexión se realiza mediante un objeto de la clase **java.sql.Connection**. La construcción típica implica indicar la URL de la base de datos, el usuario y la contraseña. Ejemplo (MySQL):

```
Connection con=DriverManager.getConnection(
  "jdbc:mysql://localhost/almacen:3306","root","mimono");
```

El método estático **getConnection** de la clase **DriverManager** es el encargado de realizar la conexión. Al crearla pueden ocurrir excepciones **SQLException** que habrá que capturar. Los fallos ocurren por que la URL está mal, la base de datos no está ejecutándose, el usuario no es el correcto, etc.

La conexión se cierra con el método **close** de la clase **Connection .**

ejecución de comandos SQL. interfaz *Statement*

El método **createStatement** de la clase **Connection**, permite obtener una variable de la interfaz **Statement** que permite ejecutar sentencias SQL sobre la base de datos. Los objetos **Statement** se crean de esta forma:

```
Statement st=con.createStatement();
//con es un objeto Connection
```

executeUpdate

Este es un método **Statement** que permite ejecutar instrucciones SQL de tipo UPDATE, INSERT o DELETE y también CREATE TABLE , DROP TABLE y otros de definición de tablas. Devuelve un entero que indica el número de filas implicadas.

Ejemplo:

```
try{
   Class.forName("com.mysql.jdbc.Driver").newInstance();
   Connection con=DriverManager.getConnection(
         "jdbc:mysql://localhost/personas:3306","root","");
   Statement st=con.createStatement();
   System.out.println(st.executeUpdate("UPDATE clientes SET"+
         "sexo='V' WHERE sexo='H'"));
}
catch (SQLException e){
   System.out.println(e.getMessage());
}
```

excuteQuery

Este método permite ejecutar una consulta SELECT. Este tipo de consultas devuelven una tabla, que en Java se representa con objetos de clase **ResultSet.** El método **next** de esta clase permite avanzar de fila, mientras que hay varios métodos *get* que permiten obtener el valor de una columna. En el caso típico, el recorrido por una consulta se hace:

```
try{
   Class.forName("com.mysql.jdbc.Driver").newInstance();
   Connection con=DriverManager.getConnection(
         "jdbc:mysql://localhost/personas:3306","root","");
   Statement st=con.createStatement();
   ResultSet rs=st.executeQuery("SELECT * FROM empleados");
   while(rs.next()){
         System.out.println(rs.getString("Nombre")+
               rs.getInt("Edad"));
   }
}
catch (SQLException e){
   do {e.printStackTrace
   } while (e.getNextException());
}
```

El método **next** permite ir al registro siguiente, devuelve **false** en el caso de que no existe un siguiente registro. Al avanzar por los registros, se pueden utilizar método ***get*** para obtener valores.

Los métodos get tienen como nombre ***get*** seguido del tipo de datos a recoger (**getString, getByte, getBigDecimal,...)**. Se les pasa entre comillas el nombre del campo a recoger (nombre según esté puesto en la base de datos) o el número de campo según el orden en el que aparezca en la tabla de la base de datos (el primer campo es el 0). Para ello hay que conocer la equivalencia entre los tipos SQL y los tipos Java:

tipo SQL	tipo Java equivalente
TINYINT	**byte**
SMALLINT	**short**
INTEGER o INT	**int**
BIGINT	**long**
NUMERIC(*m*,*n*) o DECIMAL(*m*,*n*)	**java.math.BigDecimal**
FLOAT(*n*)	**float**
REAL	**double**
CHAR(*n*)	**String**
VARCHAR(*n*)	**String**
BOOLEAN o BOOL o BIT	**boolean**
DATE	**java.sql.Date**
TIME	**java.sql.Time**
TIMESTAMP	**java.sql.Timestamp**
BINARY	**byte[]**
BLOB	**java.sql.Blob**
CLOB	**java.sql.Clob**
ARRAY	**java.sql.Array**

wasNull

Al leer un determinado campo con alguna función **get** puede ocurrir que el valor leído sea nulo. En ese caso se puede comprobar mediante la función booleana **wasNull.** Esa función devuelve **true** si la última función **get** obtuvo un valor nulo de la base de datos. Ejemplo:

```
int n=rs.getInt("Valor");
if (rs.wasNull()){
 System.out.println("Se leyó un valor nulo");
}
```

Excepciones en la base de datos

SQLException

En el paquete **java.sql** se encuentra la clase **SQLException** que captura las excepciones ocurridas en el manejo de la base de datos. Su uso no difiere del resto de excepciones, pero incorpora nuevos métodos interesantes:

- **getSQLState**. Describe el error según las convenciones XOPEN.
- **getMessage.** El método típico de excepciones, salvo que éste recoge el texto que envía el controlador JDBC según lo informado por el gestor de bases de datos, lo que le hace muy efectivo.

- **getErrorCode**. Devuelve el código de error ocurrido (según lo informado por el gestor de la base de datos)
- **getNextException.** Que permite ver la siguiente excepción ocurrida, ya que a veces ocurren varias a la vez (especialmente en transacciones y operaciones complejas).

Ejemplo:

```
try{
   //instrucciones de manejo de la base de datos
}
catch(SQLException sqle){
   while(e!=null){
         System.err.println("Estado: "+e.getSQLState());
         System.err.println("Código: "+e.getErrorCode());
         System.err.println("Mensaje: "+e.getMessage());
         e.getNextException();
   }
}
```

SQLWarning

Otras veces, ocurre que la base de datos provoca excepciones, sino advertencias (*warnings***)**. Los objetos que las provocan (sobre todo los **ResultSet)** las van almacenando en objetos de tipo **SQLWarning.** Para ver los errores basta con llamar repetidamente al método **getSQLWarning.** En cada llamada obtendremos un nuevo objeto **SQLWarning** hasta que, finalmente no haya más (devolverá entonces el valor **null**).

Los métodos que permiten observar el contenido de la advertencia son los mismos que los de la clase **SQLException,** ya que esta clase es heredera suya. Añade el método **getNextWarning** que funciona de forma similar a **getNextException,** pero que en esta clase no se suele utilizar ya que las sucesivas llamadas al método **getSQLWarnings** provocan el mismo resultado.

El método **clearWarnings** de la clase **ResultSet** permite borrar las advertencias almacenadas hasta ese momento.

resultados con posibilidades de desplazamiento y actualización. JDBC 2.0

Se ha visto anteriormente como el objeto de clase **ResultSet** es el encargado de almacenar los resultados de una consulta **SELECT** creada con el método **executeQuery** de la clase **Statement.** Este objeto se puede recorrer del primer registro al último mediante el método **next** y se puede obtener el valor de cada campo mediante métodos **get** (***getInt, getString, getByte, getBigDecimal,...)***.

En JDBC 2.0 se puede además realizar recorridos en todas las direcciones sobre los registros e incluso añadir registros. Para ello el controlador debe ser compatible con JDBC 2.0, de otro modo no se podrán utilizar estas funciones.

el método *createStatement* de la clase *Connection*

Anteriormente vimos como una consulta se crea mediante un objeto *Statement* el cual, a su vez, es creado utilizando el método *createStatement*. Sin embargo este método puede utilizar dos argumentos que permiten indicar el tipo de *ResultSet* que obtendría el *Statement*. Esos dos argumentos son el **tipo** y la **concurrencia** y permiten utilizar estas constantes.

Para el tipo:

- **ResultSet.TYPE_FORWARD_ONLY**. El conjunto de resultados no tendrá desplazamiento. Sólo se podrá utilizar el método **next** (este es el único valor para JDBC 1.0 y es el valor predeterminado si se construye el *Statement* sin argumentos).
- **ResultSet.TYPE_SCROLL_INSENSITIVE**. El conjunto de resultados tendrá desplazamiento completo pero no tendrá en cuenta los cambios.
- **ResultSet.TYPE_SCROLL_SENSITIVE**. Conjunto de resultados con desplazamiento y sensibilidad a los cambios.

Para la concurrencia:

- **ResultSet.CONCUR_READ_ONLY**. La base de datos no puede ser modificada mediante el uso del conjunto de resultados
- **ResultSet.CONCUR_UPDATABLE**. La base de datos es actualizable.

desplazamiento por los conjuntos de resultados

Si se permite el desplazamiento, la clase **ResultSet** posee los siguientes métodos para desplazarse por los registros:

método	uso
boolean next()	Avanza el puntero de registros del conjunto de resultados al siguiente registro. Devuelve **true** si existe registro siguiente.
boolean previous()	Coloca el puntero de registros en el registro anterior si lo hay, si no lo hay devuelve **false**
boolean absolute(int registro**)**	Coloca el puntero de registros en la fila indicada. Si esa fila no existe, devuelve **false**. Si el número de fila se indica con un número negativo, la fila se cuenta desde el final.

método	uso
boolean relative(int fila**)**	Coloca el puntero de registros en la fila indicada a partir de la posición actual del puntero. Si esa fila no existe, devuelve **false**. El número de fila se puede indicar de forma negativa y en ese caso el puntero se mueve hacia el primer registro (si es positivo se mueve hacia el final).
boolean first()	Coloca el puntero en el primer registro. Si no hay primer registro, devuelve **false**
boolean last()	Coloca el puntero en el último registro. Si no hay último registro, devuelve **false**
void beforeFirst()	Coloca el puntero delante del primer registro. El método **next** se movería al primer registro si se utiliza tras esta orden.
void afterLast()	Coloca el puntero detrás del último registro. El método **previous** se movería al último registro si se utiliza tras esta orden.
boolean isFirst()	Devuelve **true** si el puntero está situado en el primer registro.
boolean isLast()	Devuelve **true** si el puntero está situado en el último registro.
boolean isBeforeFirst()	Devuelve **true** si el puntero está situado delante del primer registro.
boolean isAfterLast()	Devuelve **true** si el puntero está situado detrás del último registro.
int getRow()	Obtiene el número de registro actual

modificación de datos

Los conjuntos de resultados se pueden utilizar también para modificar los datos obtenidos por la consulta SELECT (siempre y cuando sea posible). Para ello se necesitan utilizar los métodos **update** de la clase **ResultSet** que permiten modificar el contenido de un campo en la posición actual del puntero de registros.

Se trata de un conjunto de métodos que comienzan con la palabra *update* seguida del tipo de datos Java del campo y un segundo parámetro que indica el nuevo valor para el campo. Ejemplo:

```
rs.first(); //Si rs es un ResultSet, se coloca el puntero en
            //el primer registro
rs.updateDouble("Precio",2.34); //El precio valdrá 2,34 en
                                //el primer registro
rs.updateString("tipo","tr"); //El campo tipo valdrá tr
rs.updateRow();
```

El método **updateRow** es el que permite actualizar la base de datos con los nuevos cambios. Se debe utilizar cuando estamos seguros de que los cambios son los correctos. Si queremos anular los cambios se debe utilizar el método **cancelRowUpdates**

adición de datos

Para añadir un nuevo registro (una nueva fila) en el conjunto de resultados obtenido. Hay que emplear los métodos anteriores de modificación de datos (métodos *update*) sobre una fila especial conocida como **fila de inserción de registros.**

Para ello los pasos son:

1> Colocar el cursor en la fila de inserción de registros mediante el método **moveToInsertRow.**

2> Actualizar los datos de los campos de ese nuevo registros usando los métodos **update** (*updateString, updateInt, updateBigDecimal,...*).

3> Añadir el registro en la base de datos con **insertRow** (se puede cancelar con **cancelRowUpdates**)

4> Colocar el cursor en la posición anterior utilizando el método **moveToCurrentRow**.

borrar registros

Se puede borrar el registro actual del conjunto de resultados utilizando el método **deleteRow**.

actualizar registro

El método **refreshRow** del **ResultSet** actualiza el valor del registro actual, según lo que valga ahora en la base de datos. Se usa por si acaso se ha modificado el valor de los datos desde otro cliente de la base de datos.

metadatos

Hay casos en los que se requiere conocer la estructura de una base de datos (nombre y diseño de las tablas, tipos de los campos, etc.). Los datos que describen la estructura de las bases de datos es lo que se conoce como **metadatos**.

Los metadatos se obtienen utilizando el método **getMetaData** de la clase **Connection**., por lo que es el objeto de la conexión el que permite obtener estos metadatos. El resultado de este método es un objeto de clase **DatabaseMetaData**.

```
//Si con es el objeto Connection:
DatabaseMetaData metadatos=con.getMetaData();
```

métodos de la clase *DatabaseMetadata*

Una vez obtenido el objeto se pueden utilizar estos métodos para obtener información sobre la base de datos:

método de DatabaseMetadata	uso
boolean allTablesAreSelected()	Indica si se pueden utilizar todas las tablas devueltas por el método **getTables**
boolean deletesAreDetected(int tipo**)**	Devuelve **true** si las filas borradas en un conjunto de resultados (objeto **ResultSet**) pueden ser detectados por el método **rowDeleted** de la clase **ResultSet**. El parámetro tipo indica el tipo de conjunto de resultados (ResultSet.TYPE_FORWARD_ONLY, ResultSet.TYPE_SCROLL_INSENSITIVE o ResultSet.TYPE_SCROLL_SENSITIVE).
ResultSet getCatalogs()	Devuelve una tabla de resultados **ResultSet** cuyo contenido es la columna **TABLE_CAT** y donde cada fila es un registro que representa un catálogo de la base de datos.
String getCatalogSeparator()	Devuelve los caracteres que separan el nombre del catálogo respecto del nombre de una tabla.
ResultSet getColumns(String catálogo, **String** plantillaEsquema, **String** plantillaTabla, **String** plantillaCampo**)**	Obtiene una tabla de resultados (**ResultSet**) en la que cada fila es un campo que cumple el nombre de campo indicado en la plantilla de campo (que puede ser **null**). Además los campos pertenecen a la tabla indicada en el catálogo (un catálogo agrupa esquemas, puede ser **null**) y plantilla de esquema (un esquema agrupa tablas relacionadas y puede ser **null***)* señalados. El conjunto de resultados posee 22 columnas. Las fundamentales son: ◉ **TABLE_CAT**. Catálogo de la tabla. ◉ **TABLE_SCHEM**. Esquema de la tabla. ◉ **TABLE_NAME**. Nombre de la tabla ◉ **COLUMN_NAME**. Nombre del campo. ◉ **DATA_TYPE**. Un entero que indica el tipo. Se puede comparar con las constantes de la clase **java.sql.Types**. ◉ **TYPE_NAME**. Nombre del tipo de datos de la columna según las directrices del gestor de base de datos empleado ◉ **COLUMN_SIZE**. Entero que indica el tamaño del campo ◉ **DECIMAL_DIGITS**. Número de dígitos decimales.

método de DatabaseMetadata	uso
	◉ **IS_NULLABLE**. Cadena *"YES"* o *"NO"*que indica si el campo puede contener nulos.
Connection **getConnection()**	Devuelve el objeto **Connection** relacionado con este objeto
String **getDatabaseProductName()**	Devuelve el nombre comercial del sistema gestor de base de datos en uso
String **getDatabaseProductVersion()**	Devuelve la versión de producto del sistema de base de datos en uso
int **getDriverMajorVersion()**	Devuelve el número mayor de versión del driver JDBC
int **getDriverMinorVersion()**	Devuelve el número menor de versión del driver JDBC
String **getDriverName()**	Devuelve el nombre del driver JDBC en uso
String **getIdentifierQuoteString()**	Devuelve el carácter utilizado para delimitar nombres de campo o de tabla en la base de datos (normalmente la comilla simple). Si se devuelve un espacio en blanco es que no hay delimitador.
ResultSet **getImportedKeys(**String catalog, String schema, String tabla**)**	Obtiene un conjunto de resultados que describe la clave de la tabla indicada para el catálogo y esquema correspondientes (si no hay catálogo o esquema se indica **null**).
ResultSet **getIndexInfo(**String catalog, String schema, String tabla, boolean única, boolean aproximación**)**	Obtiene un conjunto de resultados que describe los índices de la tabla indicada.
int **getMaxColumnNameLength()**	Devuelve el máximo número de caracteres que puede poseer un campo en esta base de datos.
int **getMaxColumnsInGroupBy()**	Obtiene el máximo número de elementos que pueden indicarse parea esta base de datos en la cláusula GROUP BY de una sentencia SQL
int **getMaxColumnsInIndex()**	Devuelve el máximo número de campos que se pueden colocar en un índice para esta base de datos
int **getMaxColumnsInOrderBy()**	Obtiene el máximo número de elementos que pueden indicarse para esta base de datos en la cláusula ORDER BY de una sentencia SQL
int **getMaxColumnsInSelect()**	Devuelve el máximo número de elementos que pueden indicarse en una instrucción SELECT para esta base de datos
int **getMaxColumnsInTable()**	Devuelve el máximo número de campos que puede poseer una tabla de la base de datos.

método de DatabaseMetadata	uso
int **getMaxConnections()**	Obtiene el número máximo de conexiones concurrentes que pueden abrirse para esta base de datos.
int **getMaxIndexLength()**	Devuelve el máximo número de bytes que puede tener un índice de la base de datos.
int **getMaxRowSize()**	Devuelve el máximo número de bytes que puede tener un registro de la base de datos.
int **getMaxStatements()**	Obtienen el máximo número de objetos de consulta (**Statements**) que se pueden crear usando esta base de datos.
int **getMaxTableNameLength()**	Obtiene el máximo número de caracteres que puede tener el nombre de una tabla en esta base de datos.
int **getMaxTablesIn Select()**	Obtiene el máximo número de tablas que pueden intervenir en una consulta SELECT
String **getNumericFunctions()**	Obtiene una lista de las funciones internas de tipo numérico de esta base de datos separadas por comas.
ResultSet **getPrimaryKeys(**String catálogo, String esquema, String tabla**)**	Obtiene una tabla de resultados (**ResultSet**) cuyos registros representan una tabla del esquema (que puede ser **null**) y catálogo (también puede ser **null**) indicados. La tabla posee estas columnas: ◉ **TABLE_CAT**. Nombre del catálogo (puede ser **null**) ◉ **TABLE_SCHEM**. Nombre del esquema (puede ser **null**). ◉ **TABLE_NAME**. Nombre de la tabla ◉ **COLUMN_NAME**. Nombre del campo que forma parte de la clave primaria. ◉ **KEY_SEQ**. Número de secuencia (**short**) ◉ **PK_KEY**. Nombre de la clave (puede ser **null**)
String **getSQLKeywords()**	Devuelve una cadena con todos los comandos, separados por comas, SQL reconocidos por el gestor de bases de datos en uso que no son parte de SQL ANSI 92.
String **getStringFunctions()**	Obtiene una lista de las funciones internas de tipo String de esta base de datos separadas por comas.
String **getSystemFunctions()**	Obtiene una lista de las funciones internas de sistema de esta base de datos separadas por comas.

método de DatabaseMetadata	uso
ResultSet getTables(String catálogo, **String** plantillaEsquema, **String** plantillaTabla, **String** tipos[]**)**	Obtiene una tabla de resultados (*ResultSet*) en la que cada fila es una tabla del catálogo (un catálogo agrupa esquemas, puede ser *null*), plantilla de esquema (un esquema agrupa tablas relacionadas y puede ser *null)*, patrón de nombre de tabla (permite indicar nombres de tabla) y tipo indicado. El tipo se indica con un array String que puede contener como elementos: ◉ **"TABLE"**. Tabla normal. ◉ **"VIEW"**. Vista. ◉ **"SYSTEM TABLE"**. Tabla de sistema ◉ **"GLOBAL TEMPORARY"**. Tabla temporal. ◉ **"LOCAL"**. Tabla Local. Si el tipo es *null* se obtendrán todas las tablas de la base de datos. El conjunto de resultados posee cinco columnas, que son: ◉ **TABLE_CAT**. Catálogo de la tabla. ◉ **TABLE_SCHEM**. Esquema de la tabla. ◉ **TABLE_NAME**. Nombre de la tabla ◉ **TABLE_TYPE**. Tipo de tabla. ◉ **REMARKS**. Comentarios.
ResultSet getTablesTypes()	Obtiene un **ResultSet** donde cada registro es un tipo de tabla soportado por el gestor de bases de datos.
String getURL()	Obtiene una cadena con el URL del sistema gestor de bases de datos.
String getUserName()	Devuelve el nombre de usuario actual del gestor de bases de datos.
boolean isReadOnly()	Indica si la base de datos es de sólo lectura
boolean nullsAreSortedAtEnd()	Indica si los valores nulos aparecen los últimos en una columna ordenada.
boolean nullsAreSortedAtStart()	Indica si los valores nulos aparecen los primeros en una columna ordenada.

método de DatabaseMetadata	uso
boolean othersDeletesAreVisibles()	Indica si los registros borrados por otros usuarios son visibles.
boolean othersInsertsAreVisibles()	Indica si los registros insertados por otros usuarios son visibles.
boolean othersUpdatesAreVisibles()	Indica si los registros actualizados por otros usuarios son visibles.
boolean ownDeletesAreVisibles()	Indica si nuestros registros borrados son visibles.
boolean ownInsertsAreVisibles()	Indica si nuestros registros insertados son visibles.
boolean ownUpdatesAreVisibles()	Denota si nuestros registros actualizados son visibles.
boolean storesLowerCaseIdentifiers()	Indica si los identificadores de la base de datos sin delimitador son almacenados en minúsculas.
boolean storesLowerCaseQuotedIdentifiers()	Indica si los identificadores de la base de datos con delimitador son almacenados en minúsculas.
boolean storesMixedCaseIdentifiers()	Denota si los identificadores de la base de datos sin delimitador son almacenados en minúsculas o mayúsculas.
boolean storesMixedCaseQuotedIdentifiers()	Indica si los identificadores de la base de datos con delimitador son almacenados en minúsculas o mayúsculas.
boolean storesUpperCaseIdentifiers()	Indica si los identificadores de la base de datos sin delimitador son almacenados en minúsculas.
boolean storesUpperCaseQuotedIdentifiers()	Indica si los identificadores de la base de datos con delimitador son almacenados en mayúsculas.
boolean supportsAlterTableWithAddColumn()	Indica si la base de datos permite añadir columnas usando la instrucción SQL ALTER TABLE
boolean supportsAlterTableWithDropColumn()	Indica si la base de datos permite borrar columnas usando la instrucción SQL ALTER TABLE
boolean supportsANSI92FullSQL()	Denota si el SQL ANSI 92 está completamente soportado por la base de datos
boolean supportsBatchUpdates()	Indica si la base de datos admite procesos por lotes.
boolean supportsCorrelatedSubqueries()	Indica si la base de datos soporta instrucciones SELECT anidadas
boolean supportsGroupBy()	Indica si la base de datos soporta la cláusula GROUP BY en la instrucción SELECT

método de DatabaseMetadata	uso
boolean supportsLikeEscapeClause()	Indica si la base de datos soporta la cláusula LIKE en la instrucción SELECT
boolean suportsMixedCaseIdentifiers()	Indica si el nombre de los identificadores sin delimitador de la base de datos cuando combinan mayúsculas y minúsculas se almacenan de esta manera o no.
boolean supportsMixedCaseQuotedIdentifiers()	Indica si el nombre de los identificadores con delimitador de la base de datos cuando combinan mayúsculas y minúsculas se almacenan de esta manera o no.
boolean supportsOrderByUnrelated()	Denota si la cláusula ORDER BY puede utilizar campos diferentes de los enunciados en el SELECT
boolean supportsResultSetConcurrency(int tipo, **int** concurrencia**)**	Indica si la base de datos soporta la concurrencia indicada para el tipo remarcado (véase el método *createStatement* de la clase *Connection*, página 267).
boolean supportsResultSetType(int tipo**)**	Indica si los *ResultSet* obtenidos desde esta base de datos pueden ser del tipo indicado, (véase el método *createStatement* de la clase *Connection*, página 267).
boolean supportsSelectForUpdate()	Indica si el SQL de la base de datos permite instrucciones SELECT FOR UPDATE
boolean supports Transactions()	Indica si la base de datos soporta transacciones
boolean updatesAreDetected()	Indica si se pueden detectar filas actualizadas con el método **rowUpdated** de la clase **ResultSet**
boolean usesLocalFiles()	Indica si la base de datos almacena archivos locales

En este listado sólo se han señalado las instrucciones más utilizadas, hay el doble de métodos en realidad con respecto a los aquí señalados.

metadatos de una consulta

El método **getMetaData** de la clase **ResultSet** da como resultado un objeto de tipo **ResultSetMetaData** que devuelve información de control del conjunto de resultados. Sus métodos más interesantes son:

método de ResultSetMetaData	uso
int getColumnCount()	Obtiene el número de columnas del conjunto de resultados
int getColumnDisplaySize(int númeroDeColumna**)**	Indica la anchura que se destina en pantalla a mostrar el contenido de la columna indicada

método de ResultSetMetaData	uso
String **getColumnName(**int nColumna**)**	Devuelve el nombre de la columna con el número indicado
int **getColumnType(**int nColumns**)**	Obtiene el tipo de datos SQL estándar de la columna indicada
int **getColumnTypeName(**int nColumna**)**	Obtiene el tipo de datos compatible con la base de datos en uso de la columna indicada
int **getPrecission(**int nColumna**)**	Devuelve la precisión de decimales de la columna dada
int **getScale(**int nColumna**)**	Obtiene el número de decimales que se muestran de la columna
boolean **isAutoincrement(**int nColumna**)**	Indica si el campo es autoincrementable.
boolean **isCaseSensitive(**int nColumna**)**	Indica si la columna distingue entre mayúsculas y minúsculas
boolean **isReadOnly(**int nColumna)	Indica si el campo es de sólo lectura.
boolean **isSearchable(**int nColumna**)**	indica si la columna puede figurar en el apartado WHERE de una consulta SELECT
boolean **isSigned(**int nColumna**)**	Indica si la columna posee números con signo

proceso por lotes

Una mejora importante de JDBC 2 es la facultad de procesar múltiples instrucciones SQL mediante lotes (*Batch*). Los procesos por lotes permiten recopilar y lanzar una serie larga de instrucciones.

Esto se realizar mediante los métodos **addBatch** y **executeBatch** de la clase **Statement**. El primero permite añadir nuevas instrucciones al proceso por lotes. El segundo lanza las instrucciones almacenadas.

El resultado de **executeBatch** es un array de enteros donde cada elemento es el número de filas modificadas por la acción lanzada correspondiente. Es decir si el primer comando SQL del proceso modifica tres filas y el segundo seis, el resultado es un array de dos elementos donde el primero vale tres y el segundo seis.

Sólo se permiten colocar instrucciones SQL de actualización (UPDATE, INSERT, CREATE TABLE, DELETE,..). El método **clearBatch** permite borrar el contenido del proceso por lotes (de hecho borra todas las consultas del *Statement*),

Ejemplo:

```
Statement st=con.createStatement;//con es la conexión
stat.addBatch("CREATE TABLE......
stat.addBatch("INSERT INTO....
...
int cuentaFilas[]=stat.executeBatch();
```

tecnologías del lado del servidor

Internet es uno de los medios fundamentales en los que puede residir un aplicación actual. Java proporciona la posibilidad crear applets, para incorporar Java a las páginas web; de modo que es la máquina cliente que visita la página la encargada de traducir el Java inmerso en la página. Esto tiene grandes desventajas.

Uno de los problemas de la creación de aplicaciones TCP/IP del lado del cliente (como las applets por ejemplo) es que el cliente debe poseer software adaptado a esa tecnología. Si no, la aplicación no carga correctamente.

Esto ocurre con cualquier aplicación del lado del cliente. Así una simple página web HTML requiere por parte del cliente un navegador compatible con los códigos HTML originales; si se usa JavaScript, el navegador del cliente debe poseer la capacidad de interpretar código JavaScript compatible con el original; si se usan Applets el navegador debe tener instalado el plugin de Java.

El cliente puede desactivar todas estas tecnologías o incluso no tenerlas instaladas. La única solución es hacer que el cliente las instale (algo que no suele funcionar muy bien debido a la desconfianza que tiene el usuario ante esta instalación).

Por si fuera poco, los usuarios siempre van por detrás de las innovaciones tecnológicas, por lo que sus versiones de software distan de ser las últimas.

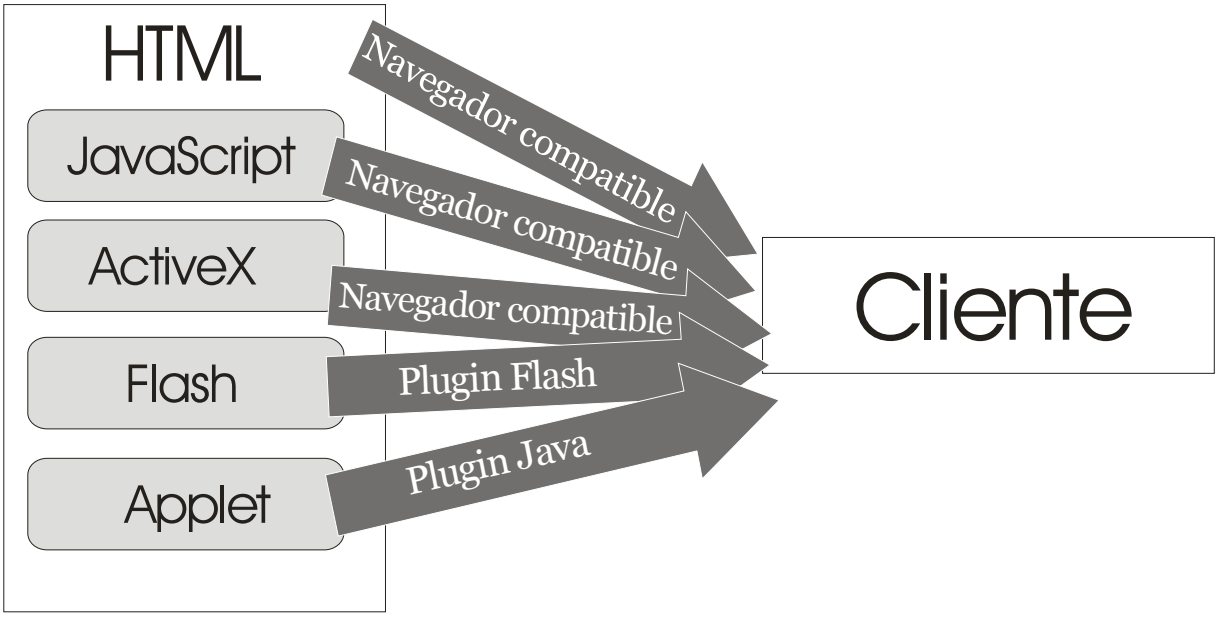


Ilustración 35,Algunas tecnologías del lado del cliente y software necesario para ellas

Para evitar estos problemas se idearon técnicas de creación de aplicaciones para la web del lado del servidor. En las que la interpretación se realiza en el propio servidor y no en el cliente. Veremos a continuación las principales.

CGI

Common Gateway Interface, o interfaz de pasarela común (CGI) es la tecnología de servidor más veterana. Apareció debido a las limitaciones de HTML para crear verdaderas aplicaciones de red.

CGI define una serie de características que permiten comunicar a una página con una aplicación residente en un servidor. La aplicación puede estar escrita casi en cualquier lenguaje (aunque el más utilizado es el lenguaje **Perl**) lo único que tiene conseguir es que su salida y entrada ha de ser pensada para comunicarse con la web de forma que el usuario no necesite ningún software adicional (los datos de salida suelen prepararse en formato HTML).

El servidor en el que reside la aplicación CGI debe tener implementado un compilador compatible con el lenguaje utilizado para escribir la aplicación.

ASP y ASP.NET

ASP parte de simplificar la idea de la tecnología de servidor. Se trata de páginas HTML que poseen etiquetas especiales (marcadas con los símbolos <% y %>) que marcan instrucciones (en diversos lenguajes, sobre todo VBScript) que debe ejecutar el servidor.

El servidor interpreta esas instrucciones y obtiene una página HTML (que es la que llega al cliente) resultado del código ASP. Es una tecnología muy exitosa gracias a la cantidad de programadores Visual Basic.

El problema es que está pensada únicamente para servidores web IIS (*Internet Information Server* los servidores web de Microsoft).

.NET es la nueva implementación de la tecnología de servidores de Microsoft que incorpora diversos lenguajes bajo una interfaz común para crear aplicaciones web en los servidores IIS. Se pueden utilizar varios tipos de lenguajes (especialmente C# y VBScript) combinados en páginas ASP.NET con directrices de servidor y posibilidad de conexión a bases de datos utilizando ADO (plataforma de conexión abierta de Microsoft para acceder a bases de datos, sucesora de ODBC).

ColdFussion

Tecnología soportada por la empresa Macromedia que parte de la misma idea que ASP, pero en lugar de usar etiquetas <%, utiliza etiquetas especiales que son traducidas por el servidor. No posee lenguaje de script lo que hace más fácil su aprendizaje, ya que sólo añade a las etiquetas normales de HTML una serie de etiquetas entendibles por los servidores de aplicaciones ColdFussion.

La interpretación de los códigos ColdFussion proporciona de nuevo una página HTML. Su desventaja es que no es muy estándar y que es difícil la creación de aplicaciones muy complejas.

PHP

Es una tecnología similar a ASP. Se trata de una página HTML que posee etiquetas especiales; en este caso son las etiquetas <?, que encierran comandos para el servidor escritos en un lenguaje script especial (es un lenguaje con muchas similitudes con Perl).

La diferencia con ASP es que es una plataforma de código abierto, compatible con Apache y que posee soporte para muchos de los gestores de base de datos más populares (Oracle, MySQL, etc.).

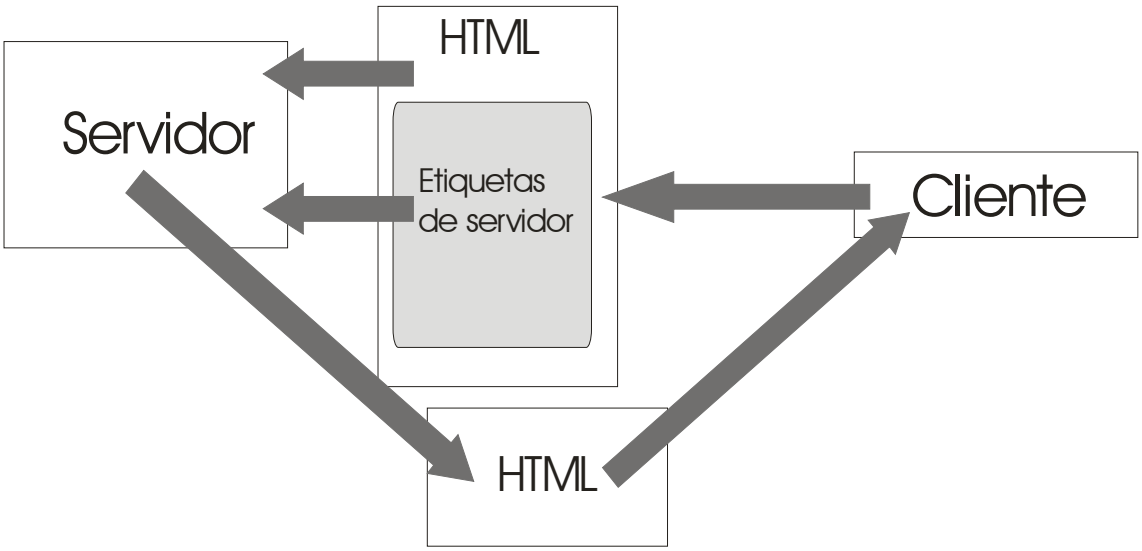


Ilustración 36, Esquema de las tecnologías de servidor. El servidor traduce las etiquetas y devuelve al cliente el código HTML resultado de esas etiquetas

Servlets y JSP

Son las tecnologías planteadas por Java para crear aplicaciones cuyas tecnologías residan en el lado del servidor. JSP es similar a ASP y PHP. Los servlets son equivalentes a las applets, pero en el lado del servidor. Ambas tecnologías se estudian en este capítulo. Las dos forman parte de lo que hoy en día se conoce como J2EE (*Java 2 Enterprise Edition*). Una aplicación JSP o Servlet se conoce como **aplicación web**

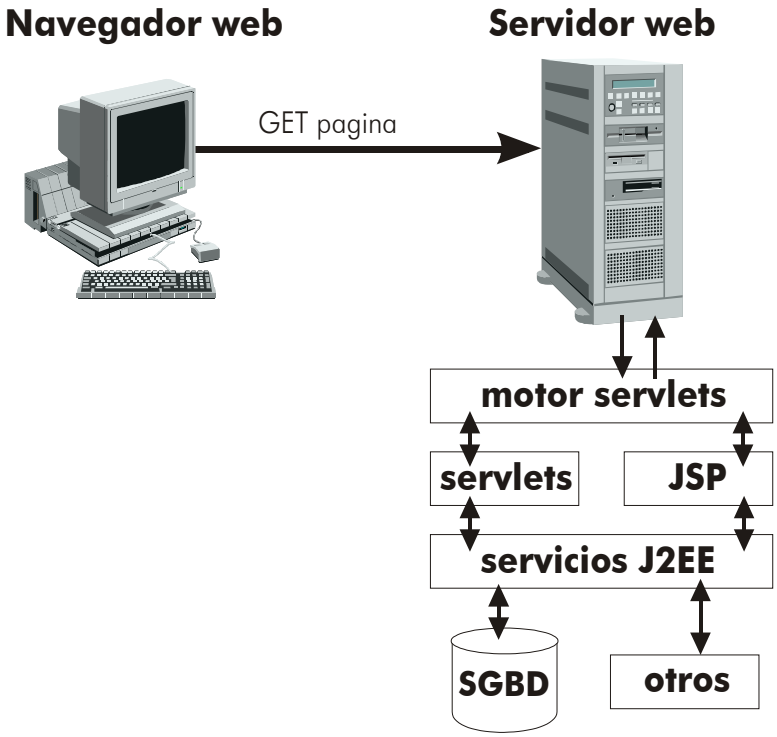


Ilustración 37, esquema de aplicación usando Servlets y JSP

J2EE

Se trata de una plataforma completa para construir aplicaciones completas desde la web basadas en el lenguaje Java. Se trata de una serie de tecnologías que permiten escribir aplicaciones en el lado del servidor para proporcionar servicios desde redes TCP/IP. Lógicamente todas estas técnicas se basan en el lenguaje Java.

Mantienen el paradigma Java de la portabilidad incluso en el caso de cambiar el sistema operativo del servidor. Sus APIs están en el paquete **javax**. Las fundamentales son:

- Servlets
- JSP
- JAXP (API de procesamiento de documentos XML)
- EJB (*Enterprise Java Beans*)

Para ello hace falta ejecutar la aplicación J2EE en servidores web compatibles que posean un servidor de aplicaciones compatibles, por ejemplo:

- WebLogic http://www.bea.com
- Inprise/Borland AppServer: http://www.inprise.com
- IBM WebSphere ApplicationServer: http://www.ibm.com/software/webservers
- IONA iPortal Application Server: http://www.iona.com
- iPlanet Application Server: http://www.iplanet.com
- Macromedia JRun Server: http://www.allaire.com
- Oracle Application Server: http://www.oracle.com
- Sun Java 2 SDK Enterprise Edition: http://java.sun.com/j2ee/j2sdkee (válido sólo para aprender)
- Sun ONE (estrategia de servidores completa de Sun)

De forma gratuita y de código abierto, *open source* (implementaciones parciales):

- Tomcat (subproyecto de Jarka): http://jakarta.apache.org/tomcat (válido para JSP y Servlets)
- JBoss: http://www.jboss.org
- Evidan JOnAS: http://www.evidian.com/jonas

empaquetamiento de las aplicaciones web

Las aplicaciones (sean servlets o JSP) se empaquetan dentro de archivos **war** (pueden ser generados por la aplicación **jar**). La estructura de estos archivos es:

- Un directorio raíz del que parten todas las carpetas de la aplicación, sólo este directorio es visible para los navegadores
- El directorio WEB-INF que contiene los archivos de ejecución de la aplicación. Este directorio es invisible a los navegadores, sólo es visible para el servidor
- La carpeta **class** donde se almacenan los archivos precompilados que forman parte de la aplicación (incluidas las Servlets)
- La carpeta **lib** en la que se almacenan los archivos **jar** para librerías de clases que utiliza la aplicación (incluidos los drivers JDBC en forma de archivo jar).
- El archivo web.xml que sirve para controlar el funcionamiento de la aplicación web.

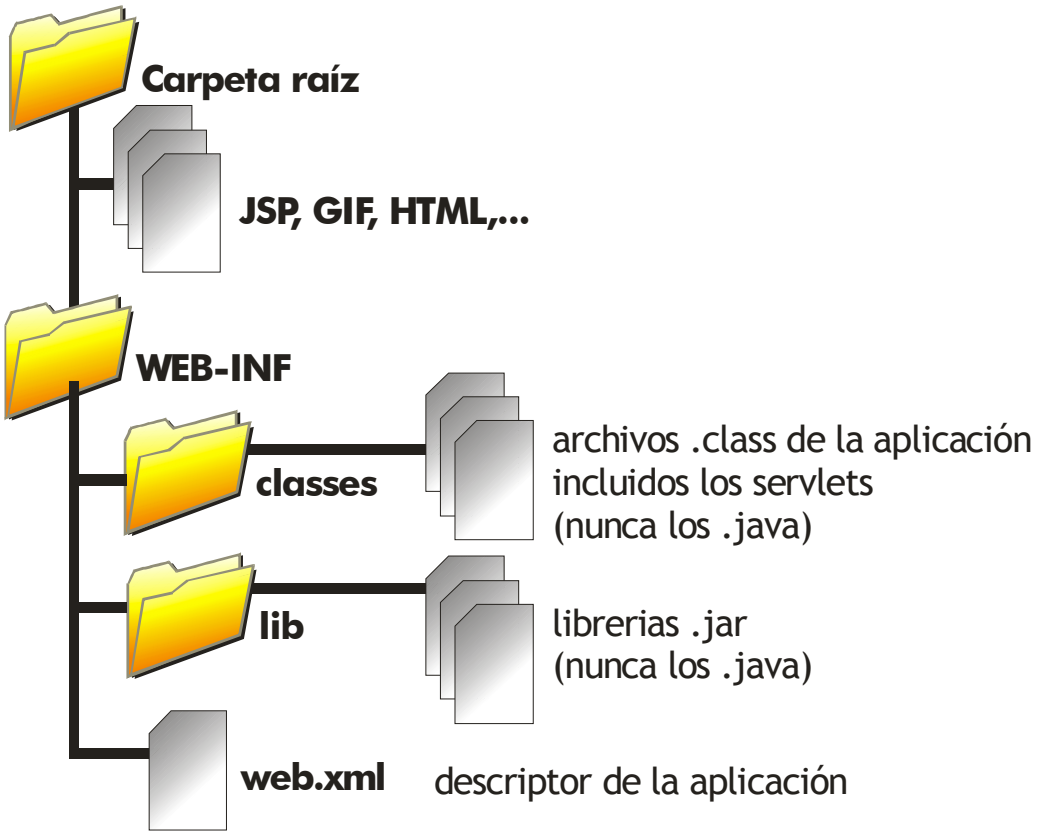


Ilustración 38, Estructura lógica de una aplicación web J2EE

Bajo esta estructura hay que colocar los archivos correspondientes.

http

El protocolo de transferencia de hipertexto es el encargado de transmitir páginas web por las redes TCP/IP. Toda la programación de aplicaciones se basa en el uso de este protocolo. Mediante http el proceso de petición y respuesta de datos sería:

1> Un cliente establece conexión por un puerto (normalmente el 80) con un servidor web. En formato de texto envía una petición

2> El servidor analiza la petición y localiza el recurso solicitado

3> El servidor envía una copia del recurso al cliente

4> El servidor cierra la conexión

Los servidores web no recuerdan ningún dato de las peticiones anteriores, cada petición es independiente. Esto supone un serio problema al programar aplicaciones mediante este protocolo. Para evitar este problema se puede hacer que el servidor nos de un número de sesión que el cliente almacenará y enviará junto a las siguientes peticiones para que el servidor recuerde.

peticiones http

Las peticiones mediante http pueden utilizar los siguientes comandos:

- **GET.** Permite obtener un recurso simple mediante su URL en el servidor
- **HEAD.** Idéntico al anterior sólo que obtiene sólo las cabeceras del recurso.
- **POST.** Petición mediante la cual se hace que el servidor acepte datos desde el cliente.
- **PUT.** Permite modificar los datos de recursos en el servidor.
- **DELETE.** Permite eliminar recursos en el servidor.
- **OPTIONS.** Petición de información sobre los métodos de petición que soporta el servidor.
- **TRACE.** Pide la petición http y las cabeceras enviadas por el cliente

datos de la petición http

Además del comando de petición, se debe indicar un segundo parámetro que es la línea de lo que se pide en el servidor. Esta línea es la ruta URL al documento pero sin las palabras http:// ni el nombre del servidor.

Es decir la URL http://www.mierv.com/index.htm, se pasa como /index.htm.

Finalmente se indica la especificación http de la petición. Puede ser HTTP/1.0 o HTTP/1.1

cabeceras de petición

Se ponen tras la línea de petición. Son líneas que incluyen una clave y su valor (separado por dos puntos). Mediante estas cabeceras el cliente indica sus capacidades e informaciones adicionales (navegador, idioma, tipo de contenido...).

ejemplo

Tras conectar por el puerto 80 con el servidor www.terra.es, la siguiente petición GET:

```
GET / HTTP/1.0
```

Devuelve la página de portada de ww.terra.es, en este formato:

```
HTTP/1.0 200 OK
Age: 14
Date: Sun, 06 Jun 2004 23:34:55 GMT
Content-Length: 38761
Content-Type: text/html
Cache-Control: max-age=15
Server: Netscape-Enterprise/4.1

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
<base href="http://www.terra.es/">
....
```

Obsérvese como primero devuelve cabeceras y finalmente el resto.

Servlets

Aparecieron en 1997 como respuesta a las aplicaciones CGI. Sus ventajas son:

- **Mejora del rendimiento.** Con las CGI lo que ocurría era que había que lanzar la aplicación con cada nueva petición de servicio. Las Servlets usan la misma aplicación y para cada petición lanzan un nuevo hilo (al estilo de los Sockets).
- **Simplicidad.** Quizá la clave de su éxito. El cliente sólo necesita un navegador http. El resto lo hace el servidor.
- **Control de sesiones.** Se pueden almacenar datos sobre las sesiones del usuario (una de las taras más importantes de http).
- **Acceso a la tecnología Java.** Lógicamente esta tecnología abre sus puertas a todas las posibilidades de Java (JDBC, Threads, etc.).

creación de Servlets

Los Servlets se deben compilar en la carpeta **classes** de la aplicación web. Se trata de una clase normal pero que deriva de **javax.servlet.Servlet**. Hay una clase llamada **javax.servlet.GenericServlets** que crea Servlets genéricos, y una clase llamada **javax.servlet.http.HttpServlet** que es la encarga de crear servlets accesibles con el protocolo http. Esos son los que nos interesan.

ciclo de vida

Un servlet genérico posee el siguiente ciclo de vida:

1> Cuando se curso la primera petición al servlet, el servidor carga el Servlet

2> Se ejecuta el método **init** del servlet

```
public void init(ServletConfig config)
throws ServletException
```

3> Cada nueva petición es manipulada por el método **service:**

```
(public void service(ServiceRequest request, ServiceResponse
response) throws ServletException, IOException).
```

request y *response* son los objetos que permiten la comunicación con el cliente.

4> El método **destroy** es llamado si se va a cerrar el servlet. Su función es liberar recursos. Llamar a destroy no corta el servlet, esto sólo lo puede realizar el servidor.

Pero para servidores web se utiliza la clase **HttpServlet** cuyo ciclo difiere un poco ya que no se utiliza el método **service** (lo sustituyen los métodos **doGet** o **doPost**). De hecho el proceso para el método **service** es :

1> El método **service(Request, Response)** heredado de **GenericServlet** transforma estos objetos en sus equivalente http

2> Se llama al nuevo método **service** que posee dos parámetros. Uno es de tipo **HttpServletRequest** (sirve para los requerimientos), el otro es un objeto de tipo **HttpServletResponse** (sirve para las respuestas).

3> El método anterior llama a **doGet(), doPost()** (que recibirán los mismos parámetros) u otro método programado, dependiendo del tipo de llamada http realizada por el cliente (si es GET se llama a doGet, si es POST se llama a doPost, etc.)

Lo normal al crear un Servlet es crear una clase derivada de **HttpServlet** y redefinir el método **doGet** o **doPost** (o ambos) dependiendo de cómo se desee recibir la información. Es más, se suele crear un método común que es llamado por doGet y doPost, ya que lo normal es que las llamadas GET o POST produzcan el mismo resultado

salida de datos desde los servlets

Tanto el método **doGet** como el método **doPost** reciben como parámetros un objeto **HttpServletResponse** y un objeto **HttpServletRequest.** El primero sirve para que el servlet pueda escribir datos. Esos datos tienen que ,necesariamente, utilizar el lenguaje HTML. Para escribir datos se utiliza el método **getWriter** de los objetos **HttpServletResponse**, el cual devuelve un objeto **PrintWriter** con el que es muy sencillo escribir (ya que posee el método **println**).

Ejemplo:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response)throws ServletException,
IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<title>Escribiendo html</title>");
...
```

el servlet *Hola Mundo*

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HolaMundo extends HttpServlet {

    public void init(ServletConfig conf)
        throws ServletException
    {
        super.init(conf);
    }

    public void doGet(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<title>Mi primer servlet</title>");
        out.println("<body>");
        out.println("<h1>Hola Mundo</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Se implementan sólo los métodos init y **doGet**. El método **doGet** (cuya definición tiene que coincidir con la del listado) es llamado si se ha requerido el servlet mediante una petición http de tipo **get**

El objeto *res* de tipo **HttpServletResponse** es el que permite el envío de datos al cliente. Necesita usar el método **setContentType** para indicar el tipo de respuesta y luego se puede obtener un **PrintWriter** cuya escritura (en formato HTML) permite pasar datos al usuario.

Tras compilar este archivo en la carpeta **class** se debe modificar el archivo **web.xml** para indicar que hay un nuevo servlet. Un posible archivo **web.xml** sería:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>Servlet_HolaMundoServlet2</servlet-name>
    <display-name>Servlet HolaMundoServlet2</display-name>
    <description>Default configuration created for
servlet.</description>
    <servlet-class>HolaMundo</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet_HolaMundoServlet2</servlet-name>
    <url-pattern>/servlet/HolaMundoServlet2</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
            30
        </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>
            index.jsp
        </welcome-file>
    <welcome-file>
            index.html
        </welcome-file>
    <welcome-file>
            index.htm
        </welcome-file>
  </welcome-file-list>
</web-app>
```

No obstante es mejor ayudarse de software que permita el relleno de este archivo.

iteración con formularios

La virtud de un servlet es el hecho de que sea dinámico, es decir, que interactúe con el usuario. Una forma muy cómoda de hacer eso es enviar parámetros al servlet. Los servlets (como los CGI y cualquier otra tecnología de servidor) pueden tomar parámetros de dos formas:

- Insertando los parámetros en una URL con apartado de consulta
- Mediante formularios en las páginas web

El primer método sólo vale para peticiones GET. Consiste en colocar tras la ruta al servlet el símbolo **?** seguido del nombre del primer parámetro, el signo '=' y el valor del parámetro (se entiende que siempre es texto). Si hay más parámetros, los parámetros se separan con el símbolo '&'. Ejemplo:

```
http://www.google.es/search?q=palencia&ie=UTF-8&hl=es&meta=
```

En este caso se llama al programa **search** y se le pasan cuatro parámetros: el parámetro **q** con valor **palencia**, el parámetro **ie** con valor **UTF-8,** el parámetro **hl** con valor **es** y el parámetro **meta** que no tiene valor alguno.

En la segunda forma se pueden utilizar peticiones GET o POST; la diferencia es que con GET se genera una dirección URL que incluye los parámetros (igual que en el ejemplo anterior), con POST los parámetros se envían de forma oculta. Los formularios son etiquetas especiales que se colocan en los documentos HTML a fin de que el usuario se pueda comunicar con una determinada aplicación. Ejemplo:

```
<HTML>
...
<FORM METHOD=GET ACTION="/servlet/ServletSaludo">
Escribe tu nombre
<INPUT TYPE=TEXT NAME=nombre SIZE=20>
<INPUT TYPE=SUBMIT VALUE="Enviar">
</FORM>
...
</HTML>
```

El método (METHOD) puede ser GET o POST, basta con indicarlo. En este código HTML, se coloca un botón de tipo **submit** (enviar) y un cuadro de texto en el que el usuario rellena sus datos. El atributo **name** indica el nombre que se le dará al parámetro que se enviará al servlet. El parámetro tendrá como valor, lo que el usuario introduzca.

El parámetro es recogido desde el Servlet por el método **getParameter** del objeto **HttpServletResponse** (se le suele llamar response a secas) de los métodos **doGet** o **doPost**:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletSaludo extends HttpServlet {
    public void init(ServletConfig conf)
        throws ServletException
    {
        super.init(conf);
    }
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<title>Mi primer servlet</title>");
        out.println("<body>");
        out.println("<h1>Hola"+
                    request.getParameter("nombre")+"</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

implementación de *SingleThreadModel*

En las conexiones a servlet se sigue un modelo de múltiples threads en el que se compartir las variables y objetos globales al servlet.

Esto significa, por ejemplo, que si se hace una conexión a una base de datos, el objeto de conexión es común a varios clientes, lo que puede provocar conflictos.

La solución es hacer que los métodos de acceso a datos compartidos sean sincronizados (**synchronized**) o implementar la interfaz **SingleThreadModel.** Esta interfaz no obliga a escribir ningún método, pero hace que el servlet atienda a un cliente cada vez (sincroniza el método **service**).

información sobre el servlet

Las clases servlet poseen un método que se puede redefinir, es el método: **public String getServletInfo().** Este método retorna una cadena que sirve para describir el servlet. En absoluto es obligatorio y sirve sólo para que algunas herramientas de gestión puedan obtener este texto.

métodos de HttpServletRequest

El objeto HttpServletRequest tiene métodos interesantes que permiten obtener información sobre:

- Parámetros de usuario (vistos anteriormente)
- Cabeceras http (pares nombre-valor)
- Otras informaciones http
- Manejo de cookies
- Obtención de la ruta de acceso de la petición
- Identificación de sesión

método	uso
String **getAuthType()**	Devuelve el nombre del sistema de autentificación del servidor (si no usa ninguno, devuelve **null**)
Cookie[] **getCookies()**	Obtiene un array con todos los objetos **Cookie** que ha enviado el cliente
String **getContentLength ()**	Devuelve el tamaño en bytes del contenido solicitado
String **getContentType ()**	Devuelve la cadena *Content Type* de la petición. El resultado es el tipo MIME que indica la petición (por ejemplo ***text/html***)
String **getContextPath()**	Devuelve la porción de la URL referida a la URL del servlet
Enumeration **getHeaderNames()**	Devuelve una enumeración de todas las cabeceras http presentes
Enumeration **getHeader(**String nombre**)**	Devuelve el contenido de la cabecera http cuyo nombre es el indicado
String **getMethod()**	Devuelve el método de llamada a la aplicación que utilizó el cliente (GET, PUT o POST por ejemplo).
String **getParameter(**String parámetro**)**	Obtiene el contenido del parámetro cuyo nombre se pasa entre comillas
Enumeration **getParameterNames()**	Obtiene una lista con los nombres de los parámetros
String[] **getParameterValues(**String nombreParámetro**)**	Devuelve un array de cadenas con los valores correspondientes al parámetro indicado o **null** si no tenía parámetro
Map **getParameterMap()**	Devuelve un objeto **map** con los valores y parámetros de la petición.
String **getPathInfo()**	Obtiene la porción de ruta de la URL que sigue al nombre del servlet

método	uso
String **getQueryString()**	Obtiene la cadena de la URL que sigue al carácter "?"
String **getRequestURL()**	Obtiene la URL completa empleada para la petición de página (incluida la zona ?)
String **getRemoteUser()**	Obtiene el nombre de usuario del cliente, si hay posibilidad de autentificarle.
HttpSession **getSession(**boolean crear**)**	Devuelve el objeto actual **HttpSession** si no hay, devuelve **null** (se creará uno nuevo si *crear* vale **true**).
boolean **isRequestedSessionIdFromCookie()**	Indica si el identificador de sesión se obtuvo de una Cookie
boolean **isRequestedSessionIdValid()**	**true** si el indicador de sesión es válido
boolean **isUserInRole(**String **rol)**	Devuelve **true** si el usuario está asociado al rol indicado.

métodos de HttpServletResponse

Permite enviar información al cliente. En los servlets http, esa información se pasa en formato HTML. Para ello se usa el método **setContentType** con valor **text/html** (aunque podría poseer otro valor).

método	uso
void **addCookie(Cookie** cookie**)**	Añade una cabecera **Set-Cookie** para la cookie indicada.
void **addHeader(**String nombre, String valor**)**	Establece el valor indicado para la cabecera http *nombre*.
void **addIntHeader(**String nombre, int valor**)**	Establece el valor indicado para la cabecera http *nombre*. El valor se coloca con valor **int**
boolean **containsHeader(**String nombre**)**	Indica si la salida ya posee la cabecera indicada
String **encodeRedirectURL(**String url**)**	Soporta el seguimiento de sesiones utilizando el identificador de sesión como parámetro de URL que se enviará mediante **sendRedirect().** Si el cliente soporta cookies, esto no es necesario.
String **getContentType()**	Obtiene la cadena MIME con el tipo de contenido de la respuesta
void **sendError(**int estado**)** throws IOException	Establece el código de estado http en el valor indicado.
void **sendError(**int estado, String msg**)** throws IOException	Lo mismo, pero además fija el mensaje de estado especificado.
void **sendRedirect(**String location**)** throws IOException	Coloca el estado http en 302 (movido provisionalmente) y se lanza al navegador a la nueva localización.

método	uso
void setCharacterSet(String tipoMIME**)**	Indica que codificación se envía en las respuestas. La codificación se debe indicar en formato estándar (el definido por la IANA), por ejemplo: text/html;charset=UTF-8 Codificación de Estados Unidos
void setContentType(String tipoMIME**)**	Establece el tipo de contenido que se enviará en la respuesta
void setHeader(String nombre, **String** valor**)**	Coloca el valor indicado a la propiedad http cuyo nombre se indica
void setIntHeader(String nombre, **int** valor**)**	Coloca el valor indicado a la propiedad http cuyo nombre se indica. El valor se coloca como **int**
void setStatus(int código**)**	Establece el código de respuesta, sólo se debe utilizar si la respuesta no es un error, sino, se debe utilizar **sendError**

Para los códigos de respuesta utilizados por varias funciones (setStatus y sendError por ejemplo), esta interfaz define una serie de constantes estática que empiezan por la palabra SC. El valor de las constantes se corresponde con el valor estándar de respuesta. Por ejemplo la constante **HttpServletResponse.SC_NOT_FOUND** vale 404 (código de error de "página no encontrada").

contexto del Servlet

Interfaz proporcionada por el servidor de aplicaciones para entregar servicios a una aplicación web. Se obtiene el objeto **ServletContext** mediante el método **getServletContext** del objeto **ServletConfig**. Este objeto se puede obtener en el método **init**, ya que lo recibe como parámetro. Mediante el contexto se accede a información interesante como:

- Atributos cuyos valores persisten entre invocaciones al servlet
- Capacidades del servlet (posibilidad de acceder a bases de datos, lectura de ficheros, etc.)
- Peticiones a otros servlets
- Mensajes de error y de otros tipos

método	uso
Object getAttribute(String nombre**)**	Obtiene el atributo del servidor de aplicaciones cuyo nombre sea el indicado. Se necesita conocer el servidor de aplicaciones para saber qué atributos posee.
Enumeration getAttributeNames()	Obtiene una enumeración con todos los atributos del contexto de servlet

método	uso
void setAttribute(String nombre, **Object** valor**)**	Establece el atributo de servidor con nombre indicado dándole un determinado valor.
ServletContext getContext(String ruta**)**	Obtiene el contexto de servlet del servlet cuya ruta se indica. La ruta debe empezar con el símbolo "/" (es decir, debe de ser absoluta) y el servlet debe estar en el mismo servidor.
int getMajorVersion()	Devuelve el número mayor de la versión de Servlet que el servidor de aplicaciones es capaz de ejecutar.
int getMinorVersion()	Devuelve el número menor de la versión de Servlet que el servidor de aplicaciones es capaz de ejecutar.
String getMimeType(String archivo**)**	Obtiene el tipo MIME del archivo cuya ruta en el servidor se indica
String getRealPath(String ruta**)**	Obtiene la ruta absoluta de la ruta interna al servidor que se indica
URL getURL(String recurso**)**	Devuelve un objeto URL correspondiente a la ruta de recurso en el servidor indicada
String getServerInfo()	Obtiene el nombre del servidor de aplicaciones que se utiliza como motor de Servlet
void removeAttribute(String nombre**)**	Elimina el atributo indicado del contexto del servlet.

sesiones http

La navegación mediante el protocolo http, dista mucho de parecerse a una comunicación cliente—servidor típica. Cuando el cliente pide un recurso, el servidor se lo da y punto. Los navegadores hacen nuevas peticiones para cada elemento de la página que haya que descargar. Si el usuario hace clic se hace una petición para el enlace.

En definitiva, el servidor se olvida del cliente en cuanto resuelve su petición. Pero esto provoca problemas cuando se desea ir almacenando información sobre el cliente (listas de productos elegidos, datos del usuario, etc.).

Para recordar datos de usuario hay varias técnicas:

- **Cookies.** Un archivo que se graba en el ordenador del cliente con información sobre el mismo.
- **Añadir un ID de sesión como parámetro.** Se añade este número bien como parámetro normal en la dirección (GET) o como parámetro oculto (POST). Lo malo es que el número de sesión hay que añadirle continuamente mientras el usuario navegue.

La interfaz **HttpSession** permite utilizar un número de sesión, que el cliente guarda y utiliza en posteriores peticiones. Se puede utilizar un manejador de sesiones utilizando el método **getSession** del objeto **HttpServletRequest** (request) utilizado. Se pueden

utilizar los métodos **getAttribute** para obtener objetos asociados al usuario con esa sesión y **setAttribute** para almacenar valores.

método	uso
Object getAttribute(String nombre**)**	Obtiene un atributo para la sesión actual. Si el nombre del atributo no existe, devuelve **null**
Enumeration getAttributeNames()	Obtiene una enumeración con todos los atributos de la sesión
void setAttribute(String nombre, **Object** valor**)**	Establece un atributo para la sesión con el nombre indicado, al que se asociará un determinado valor.
void removeAttribute(String nombre**)**	Elimina el atributo indicado de la sesión
String getID()	Obtiene el identificador de la sesión
int setMaxInactiveInterval(int segundos**)**	Establece el número de segundos que la sesión podrá estar sin que el cliente efectúe ninguna operación.
int getMaxInactiveInterval()	Obtiene el valor actual del intervalo comentado antes
long getLastAccessedTime()	Obtiene la fecha y hora en la que el usuario realizó su última petición. Devuelve el número de milisegundos de esa fecha, el formato es el mismo que el de la clase **Date**
void invalidate()	Anula la sesión actual, eliminando todos los objetos relacionados
boolean isNew()	Con **true** indica que el usuario aún no ha establecido sesión.

JSP

Se trata de una idea que puede convivir perfectamente con los servlets (de hecho refuerza esa tecnología), pero cuyo método de trabajo es distinto.

Una página JSP es una página web normal (sólo que con extensión **.jsp**) a la que se la puede añadir código java utilizando unas etiquetas especiales dentro del código de la página. Estas etiquetas son traducidas por el servidor de aplicaciones al igual que traduce el código de un servlet. Las etiquetas JSP comienzan por <% y terminan por %>

Es una técnica cada vez más popular ya que posee la potencia de Java (al mismo nivel que los servlets), pero con la gracia de escribir directamente el código en una página web, lo que facilita su diseño. Su compatibilidad es mucho mayor y su creación es más simple.

Realmente en la práctica todo JSP se convierte en un servlet, por lo que se necesita el mismo sistema de archivos (carpeta WEB-INF, web.xml, classes, lib, etc.). Por lo que las páginas JSP se almacenan en la raíz de la aplicación y sus librerías comprimidas como jar en la carpeta **lib** (todas las librerías deben almacenarse ahí, las del kit de Java no porque el servidor de aplicaciones ya las incorporará).

ciclo de vida de una página JSP

Toda página JSP atraviesa una serie de etapas:

1> Una página JSP comienza con el código nativo HTML-JSP. Es el código escrito por el programador, mezcla de HTML clásico e instrucciones Java.

2> La página JSP se transforma en el Servlet equivalente. Esto ocurre tras la petición de la página por parte del cliente. Se genera pues un archivo **class** dispuesto a ser ejecutado cuando haga falta, el servidor lo almacena para su posterior uso, por lo que en peticiones siguientes ya no se compilará: a no ser que se detecten cambios en el código original

3> Se carga la clase para cada petición entrante, el **ejemplar**, que gestiona la petición http del cliente concreto.

4> Finalmente la ejecución del servlet da lugar al código HTML que es el que recibe el cliente.

componentes de las páginas JSP

directivas

Instrucciones dirigidas al servidor web que contiene la página indicando qué tipo de código se ha de generar. Formato:

```
<%@ nombreDirectiva atributo1="valor" atributo2="valor" %>
```

page

Es la directiva más importante. Permite usar diversos atributos muy importantes. Además se puede utilizar varias etiquetas **page** en el mismo archivo. Sus atributos más importantes son:

- **import**. Lista de uno o más paquetes de clases. Los paquetes **java.lang.***, **java.servlet.***, **java.servlet.jsp.*** y **java.servlet.http.*** se incluyen automáticamente.

  Si se incluye más de un paquete, se deben de separar con comas. Además se pueden colocar varias directivas **page** con el atributo **import** (es la única directiva que se puede repetir). Ejemplo:

```
<%@ page import="java.io.*, java.sql.*, java.util.*">
```

- **contentType.** Especifica el tipo MIME de la página (normalmente **text/html**) y, opcionalmente su codificación (**"text/html; charset=iso-8859-1"**) es lo que se suele indicar en el caso de Europa occidental. Ejemplo:

```
<%@ page contentType="text/html;ISO-8859-1">
```

- **pageEncoding.** Indica la codificación de caracteres de la página (el ISO-8859-1 del ejemplo anterior). No es necesaria si se utiliza el formato completo del **ContentType**.

- **errorPage.** Permite pasar el control a otra página cuando se genera en el código una excepción sin capturar. Normalmente cuando esto ocurre, es el servidor de aplicaciones el que saca la página de error. De este modo se personalizan los mensajes de error. Las páginas de error deben colocar la directiva **page** con atributo **isErrorPage** a **true.**

  La página de error puede obtener información sobre el error ocurrido mediante la variable implícita **exception.** Esta variable está muy relacionada con los objetos **Exception** del Java tradicional y por ello posee los métodos **getMessage** y **printStacktrace** para acceder a la información de la excepción y así manipularla.

- **isErrorPage.** En caso de ser **true** indica que esta página es la versión de error de otra.

- **autoflush.** En valor **true** indica que el *buffer* de escritura se vacía automáticamente (valor por defecto).

- **buffer.** Se utiliza en combinación con el anterior para indicar la forma de enviar datos del Servlet. Se puede indicar el texto **no** que indica que no se usa buffer, y por tanto los datos se envían tan pronto son generados (si el **autoflush** está a **true**).

  Indicando un tamaño (que indicará bytes) hace que se almacenen en el buffer hasta que se llena (con autoflush a true tan pronto se llena el buffer, se envía al cliente; con valor false en el autoflush se genera un excepción si se desborda el buffer).

- **info.** Equivalente al resultado del método **getServletInfo** de la página. Permite indicar un texto para describir el archivo JSP.

- **isThreadSafe. true** si la página acepta varias peticiones simultáneas. En **false** el servlet generado está ante la situación provocada por la interfaz **SingleThreadModle** vista en el apartado de Servlets.

- **language.** Indica el lenguaje de programación utilizado para programar en la página JSP. De forma predeterminada y recomendable se utiliza el valor **Java**. Hay servidor que admiten que el lenguaje sea **JavaScript** e incluso otros, pero su portabilidad quedaría en entrdicho.

- **extends.** Permite indicar una clase padre a la página JSP. Esto sólo debería hacerse si se posee una clase padre con una funcionalidad muy probada y eficiente, ya que todos los servidores de aplicaciones proporcionan una clase padre por defecto los suficientemente poderosa para evitar esta sentencia.

  En cualquier caso la clase padre debe implementar la interfaz java.servlet.jsp.**HttpJspPage**, que define diversos métodos que, obligatoriamente, habría que definir en la clase padre.

include

La directiva **include**, al estilo de la directiva #include del lenguaje C, permite añadir a la página código incluido en otro archivo cuyo nombre se indica. Pueden haber varias directivas **include**.

```
<%@ include file="/cabecera.html" %>
```

taglib

Permite utilizar etiquetas personales cuya definición se encuentre en un descriptor de etiquetas (archivo TLD) cuya ruta se indica. Hay un segundo parámetro llamado **prefix** que permite elegir que prefijo poseen esas etiquetas. Ejemplo:

```
<%@ taglib uri="/descrip/misEtiquetas.tld" prefix="jsa" %>
```

Una etiqueta personal en el código sería:

```
<jsa:banderola>texto</jsa:banderola>
```

comentarios

Hay dos tipos:

- **Propios de JSP**. Comienzan por **<%--** y terminan por **--%>.** Sólo son visibles en el código original JSP
- **Propios de HTML.** Comienzan por <!-- y terminan por --> Son visibles en el código HTML generado por el servidor

expresiones

Comienzan por **<%=** y terminan por **%>** Entre medias se coloca una expresión Java válida que será traducida por el servidor como una instrucción **out.print** donde **out** es el objeto de salida de texto para escribir código HTML hacia el cliente. Es decir, lo que se coloca como expresión es directamente traducible como HTML

Ejemplo (hola mundo):

```
<%@page contentType="text/html"%>
<html>
<head><title>JSP Page</title></head>
<body>
    <h1>
    <%= "Hola mundo" %>
    </h1>
</body>
</html>
```

El resultado es:

```
<html>
```

```
<head><title>JSP Page</title></head>
<body>
    <h1>
    Hola mundo
    </h1>
</body>
</html>
```

Normalmente se utilizan para escribir el resultado de un método o de una variable del código Java.

```
    <p>Su nombre de usuario es
    <%= obtenerNombre() %>
    </p>
```

instrucciones

También llamadas *scriptlets,* van entre **<%** y **%>** y son sentencias puras Java. El servidor las codifica como parte del método **service** del servlet resultante. Permite escribir Java puro, utilizar objetos implícitos de JSP y variables, métodos y clases declaradas.

declaraciones

Van entre **<%!** y **%>**y sirven para declarar variables de instancia, métodos o clases internas. **No pueden utilizar objetos implícitos.**

Las variables declaradas son diferentes para cada instancia (al igual que los métodos). En definitiva, los métodos, variables o clases de las declaraciones son globales a todo el archivo JSP. Si se desea una variable o método común a todas las instancias de la página (es decir, común a todas las sesiones del archivo), entonces se pueden declarar con **static.** Ejemplo:

```
<%!
  public int factorial(int n) {
        int resultado=1;
        for(int i=1; i<=n; i++) resultado*=n;
        return resultado;
  }
%>
```

Hay que recordar que las variables declaradas en esta zona, se consideran variables globales a toda la clase del archivo JSP, es decir serán propiedades del Servlet que se genera a partir del JSP.

En las declaraciones se pueden definir métodos que luego serán utilizados dentro del JSP.

Ejemplo:

```
<%!
  public double factorial(int n){
        double resp=1;
        while(n>1) resp*=n--;
        return resp;
  }
%>
```

objetos implícitos

Se necesitan algunos objetos predefinidos para poder realizar algunas operaciones complejas.

request

Representa el objeto **HttpServletRequest** de los servlets, necesario para obtener información. Por ejemplo **request.getParameter("nombre")** recoge este parámetro para su uso posterior. Sus métodos son los comentados en la clase **HttpServletRequest** página 289.

response

Representa el objeto **HttpServletResponse** de los servlets. Permite escribir datos en la página. Sus métodos se comentaron en la página 290

pageContext

Para obtener datos sobre el contexto de la página.

session

Objeto **HttpSession.** Sus métodos se comentaron enla página 293

application

Objeto de contexto del servlet (**ServletContext**). Sus métodos son los explicados para los Servlets en la página 291

out

Representa el flujo de salida hacia HTML. Equivalente a lo que devuelve el método **getWriter** de la clase **HttpServletResponse.**

config

Objeto **ServletConfig** de esta aplicación

page

Referencia a la página JSP (es un objeto **HttpServlet**).

exception

Para captura de errores sólo válido en páginas de errores.

colaboración entre Servlets y/o JSPs

Uno de los principales problemas en el manejo de Servlets y JSPs, es el hecho de que a veces sea necesario que un servlet (o una página JSP) llame a otro Servlet o JSP, pero haciendo que la respuesta de este último depende de una condición del primero. Es decir un Servlet delega la respuesta a una página a otro Servlet.

Hay tres métodos para hacer ésta colaboración:

1> **Encadenado**. Se utilizaba al principio, antes de la aparición de J2EE, y su manejo es complejo e insuficiente para la mayoría de los problemas.

2> **Lanzamiento de solicitudes**. Permite lanzar una petición a otro Servlet o JSP. Desde la versión 2.2 del API de Servlets se utiliza un enfoque nuevo, sustituto del ya obsoleto método **getServlet** que obtenía una instancia a un Servlet. Actualmente este método siempre devuelve **null** y será eliminado en futuras versiones de Java.

El lanzamiento de solicitudes permite que un Servlet (o un JSP) llame a otro recurso utilizando los objetos **request** y **response** ya creados. Una interfaz llamada **RequestDispatcher** permite esta posibilidad

3> **Petición desde JavaScript tipo GET.** Ésta es la menos elegante, pero soluciona fácilmente pequeños problema. La instrucción JavaScript **location=*URL*** permite modificar la página web actual sustituyéndola por la página indicada por su URL. Si en esa URL se incluyen parámetros (por ejemplo (?nombre=pepe&edad=28&nacionalidad=francia), entonces resulta que se llamará al Servlet o JSP indicados pasando esos parámetros.

La instrucción desde un Servlet sería:

```
out.println("<script language='JavaScript'>"+
  "location='/servlet/form?nombre=pepe&edad=28"+
  "&nacionalidad=francia';</script>");
```

interfaz RequestDispatcher

Permite referenciar a un recurso web para recibir peticiones desde un Servlet. Permite dos métodos:

- **public void forward(**ServletRequest request, ServletResponse response**)** **throws** ServletException, IOException

  Permite enviar una solicitud a otro Servlet o página JSP. Utiliza los objetos request y response. De modo que estos objetos poseerán los datos (por ejemplo los parámetros) tal cual llegaron al Servlet.

- **public void include(**ServletRequest request, ServletResponse response**)** **throws** ServletException, IOException

  Permite incluir en el Servlet actual, la respuesta producida por otro Servlet o JSP.

El objeto RequestDispatcher necesario para poder realizar peticiones a otros recursos, se obtiene utilizando el método **getRequestDispatcher** del objeto **request.** Este

método requiere una cadena con la dirección del recurso llamado. Esa dirección parte desde el contexto raíz. Es decir si estamos en el Servlet **/apli/servicios/servicio1** y queremos obtener el servlet **/aplic/servicios/servicio2** la cadena a utilizar es "**/servicios/servicio2**".

Finalmente para poder pasar valores de un Servlet (o JSP) a otro, se puede utilizar el método **setAttribute**. Este método utiliza dos parámetros: el primero es el nombre del atributo que se desea pasar (es una cadena), el segundo es un objeto que permite colocar valor al atributo.

El Servlet (o JSP) destinatario de la petición puede obtener el valor del atributo utilizando el método **getAttribute**, que tiene como único parámetro el nombre del atributo que se desea obtener.

Los nombres de atributos cumplen las mismas reglas que los paquetes (por ejemplo un nombre de atributo sería **com.miEmpresa.nombre**). Los nombres que comienzan por **java.** o **javax.** y **sun.** ya están cogidos.

proceso de lanzamiento de peticiones

1> Desde el Servlet o página JSP, utilizar el objeto **request** para invocar al método **getRequestDispatcher**. A este método se le pasa la ruta del Servlet/JSP que será invocado

2> Configurar el objeto **request** para su uso en el Servlet o JSP invocado. En especial se suelen configurar atributos mediante la función **setAttribute** de los objetos **request.**

3> Utilizar el método **forward** de la clase **RequestDispatcher.** Este método llama al Servlet, JSP o página HTML referida en el método **getDispatcherRequest** pasando como parámetros los objetos **request** y **response**, de los que podrá obtener parámetros, atributos y cualquier otra propiedad

establecer sesiones

Se dice que el protocolo http es un protocolo sin estado, ya que las llamadas a este protocolo son independientes unas de otras. Los protocolos con estado permiten que las llamadas sean dependientes unas de otras. Esto es fundamental para realizar multitud de acciones que sería imposibles si el estado es independiente.

Temas como venta en línea, transacciones comerciales o páginas que se adapten a los criterios del usuario, no se podrían realizar sin conseguir una dependencia entre las llamadas. Por ello necesitamos establecer una **sesión.** Una sesión es una serie de solicitudes que forman una tarea completa de trabajo en la que se distinga a un cliente de otro.

El estado se consigue haciendo que el servidor recuerde información relacionada con las sesiones anteriores. Como http cierra la conexión tras resolver la solicitud, no parece posible realizar estas operaciones. Para resolver este dilema, en el API de los Servlets (y por tanto en los JSP) se han incluido herramientas que solucionan el problema. Estas son:

- **Reescritura de URLs.** Se utiliza un identificador que se añade a la dirección URL utilizada en la petición http. Cuando el cliente utiliza este identificador, el servidor le recoge para poder transmitirle de nuevo.

  Por ejemplo una llamada *http://www.miserver.com/servlet1;jsessionid=278282* generaría ese número de sesión. *jsessionid* es el la forma Java de indicar el número de sesión en la URL.

- **Campos ocultos de formulario.** Muy semejante al anterior, salvo que en lugar de rescribir el código, el servidor utiliza campos de formulario ocultos para permitir capturar el identificador de sesión. Los servlets no utilizan este enfoque

- **Cookies.** Son datos que se intercambian en la lectura y escritura y que se almacenan en el ordenador del cliente. Es un texto que el servidor le envía al cliente junto con la petición y que éste almacena en su ordenador. En cada petición el cliente enviará al servidor el cookie.

- **SSL, *Secure Socket Layer*.** Utiliza una tecnología de cifrado sobre TCP/IP (especialmente bajo http). El protocolo HTTPS se define con esta tecnología. Ser generan claves cifradas entre cliente y servidor llamadas claves de sesión.

sesiones con el API Java Servlet

La interfaz **javax.servlet.http.HttpSession** es la encargada de gestionar las sesiones. Para obtener una variable que maneje esta interfaz, se puede hacer uso del método **getSession** del objeto **request.** Sin parámetros este método obtiene un número de sesión o, si no hay sesión, establece una. Hay otra versión del método que admite un parámetro que, con valor **false,** no establece la nueva sesión si ésta no existía.

En las páginas JSP el objeto implícito **session** permite acceder a la sesión actual (es un objeto de tipo **HttpSession**).Si no se desearán registrar sesiones, entonces habría que usar la directiva: `<%@ page session=false%>`

Los métodos de la interfaz HttpSession, han sido comentados en la página 292. Los pasos para utilizar sesiones suelen ser:

1> La sesión se obtiene mediante el método **getSession** del objeto **request** (en JSP ya hay creado un objeto llamado **session**). Realmente lo que ocurre es que se creará

2> Se pueden utilizar los métodos informativos de la clase **HttpSession** para obtener información de la sesión (**getId, getCreationTime, getLastAccessedTime, isNew,**...)

3> Se utiliza el método **setAttribute** para establecer atributos de la sesión y **getAttribute** para recuperar atributos de la sesión. Esto permite grabar información en la sesión y recuperarla mientras la sesión esté activa.

4> El método **removeAttribute** permite eliminar un atributo.

5> La sesión se puede destruir de dos formas:

- La llamada al método **invalidate** de la sesión; que elimina la sesión actual.

- El cliente sobrepasa el límite de tiempo de inactividad. El intervalo de espera máxima se establece en el archivo de despliegue **web.xml** de la aplicación web:

```
<web-app>
...
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
...
</web-app>
```

El intervalo se establece en minutos (en el ejemplo el tiempo de espera máximo será de 30 minutos).

Hay que tener en cuenta que **normalmente esta gestión se realiza por cookies**. Pero esto causa un problema: qué ocurre si el usuario desactiva las cookies**.** En ese caso se podría detectar y avisar (habría que comprobar si se graban los datos o no), o mejor utilizar paso de datos sin usar cookies.

Esto se consigue utilizando el método uso de sesiones por URL. Para ello al llamar a las direcciones, en lugar de poner su URL sin más, hay que utilizar el método **encodeURL** del objeto **response**. Ese método recibe una dirección URL y devuelve la URL incluyendo la sesión. Ejemplo:

```
out.println("<A HREF="/app1/prueba.jsp">.....");
```

El código anterior colocaría en la página un enlace a la dirección **/app1/prueba.jsp.** Si el navegador acepta cookies, además grabará los datos de la sesión.

```
out.println("<A HREF="+response.encodeURL(/app1/prueba,jsp"+
">.....");
```

Este código hace lo mismo pero incluye en la URL la sesión, por lo que se permitirá usar los datos de la sesión haya cookies o no.

eventos de Sesión

Las sesiones pueden producir eventos que pueden ser escuchados mediante clases que implementen alguna de estas interfaces.

interfaz HttpSessionListener

Sirve para escuchar eventos de sesión. Ocurren estos eventos en la creación o en la destrucción (o invalidación) de la sesión. Los métodos que define son:

- **void sessionCreated(HttpSessionEvent e)**. Ocurre cuando se crea una nueva sesión en la aplicación.
- **void sessionDestroyed(HttpSessionEvent e)**. Ocurre cuando se elimina una sesión.

Cualquier objeto podría implementar esta interfaz, pero se necesita indicar este objeto en el archivo **web.xml**

```
<web-app>
...
<listener>
  <listener-class>CLaseEscuchadora</listener-class>
</listener>
...
</web-app>
```

La clase escuchadora es la encargada de gestionar las acciones necesarias en la creación o eliminación de una sesión.

interfaz HttpSessionActivationListener

Sirve para escuchar eventos de activación de la sesión. Los métodos que define son:

- **void sessionDidActivate(HttpSessionEvent e)**. Ocurre cuando la sesión se activa.
- **void sessionWillPassivate(HttpSessionEvent e)**. Ocurre cuando la sesión pasa a estado de *pasiva* (no activa)

clase HttpSessionEvent

Representa los eventos de sesión. Deriva de la clase **java.util.EventObject** a la que añade el método **getSession** que obtiene el objeto **HttpSession** que lanzó el evento.

interfaz HttpSessionBindingListener

Sirve para escuchar eventos de adición o eliminación de atributos en la sesión. Métodos:

- **void valueBound(HttpSessionBindingEvent e)**. Ocurre cuando se está añadiendo un atributo
- void **valueUnbound**(HttpSessionBindingEvent e). Ocurre cuando un atributo se está eliminando de la sesión.

Son los objetos que se utilizan como atributos de la sesión los que pueden implementar esta interfaz.

interfaz HttpSessionAttributeListener

Similar a la anterior. Permite escuchar cuándo el estado de la sesión cambia.

- **void attributeAdded(HttpSessionBindingEvent e)**. Ocurre cuando se ha añadido un atributo a una sesión
- **void attributeRemoved**(**HttpSessionBindingEvent** e). Ocurre cuando un atributo es eliminado de la sesión.

- **void attributeReplaced(HttpSessionBindingEvent** e). Ocurre cuando se reemplazó un atributo por otro. Esto sucede si se utiliza **setAttirbute** con el mismo atributo pero diferente valor.

clase HttpSessionBindingEvent

Clase que define los objetos capturados por las dos interfaces anteriores. Deriva de **java.util.EventObject** y añade estos métodos:

- **String getName()**. Devuelve el nombre del atributo implicado en el evento.
- **Object getValue()**. Devuelve el nombre del objeto implicado en el evento.
- **HttpSession getSession()**. Devuelve el nombre de la sesión relacionada con el evento.

JavaBeans

introducción

componentes

Uno de los paradigmas de la programación es aprovechar el código que ya está creado. Inicialmente la primera idea fue crear módulos y funciones que luego se aprovechan en otras aplicaciones. A esto se le llama reutilizar código. Empezó con un mero *copiar* y *pegar*, para mejorar haciendo que las funciones se agruparan en librerías que se invocaban desde el código.

Se mejoró aún más la idea con la llegada de la programación orientada a objetos. Las clases mediante las que se definen objetos encapsulan métodos y propiedades, y esto posibilita diseñar aplicaciones más fácilmente. Las clases se pueden utilizar en distintas aplicaciones. Las clases también se agrupan en librerías (o paquetes como en el caso de Java).

Pero desde hace años, el aprovechamiento del código ha sufrido una notable mejoría con la llegada de los llamados lenguajes visuales (como Visual Basic y Delphi). En estos lenguajes, se pueden colocar objetos en el código simplemente *pintándolos* en un área visual. En estos lenguajes se diseña el formulario de la aplicación y se arrastran los distintos componentes desde un cuadro de herramientas. Se modifican las propiedades de los componentes y finalmente se añade el código necesario (que será muy poco).

Como desventaja, estos lenguajes se abstraen tanto del código que no son adecuados para resolver algunos problemas (que Java sí es perfectamente capaz de resolver). Realmente es una solución ideal para la programación de elementos visuales.

A este tipo de elementos que funcionan como bloques de construcción de aplicaciones, es a los que se les llama **componentes.** De este modo una aplicación no es más que un conjunto de componentes funcionando conjuntamente.

Los componentes representan desde cosas tan sencillas como un botón a cosas más complicadas como un procesador de textos. La idea además es incluso hacer que este modelo funcione para cualquier plataforma de ordenador y para cualquier lenguaje. Esto último no se ha llegado a conseguir del todo.

Tecnologías como CORBA otorgan un modelo de componentes independiente del lenguaje gracias a una capa que hace de interfaz entre el lenguaje que accede al componente y el servidor CORBA que proporciona el componente.

Todos los componentes deben seguir un modelo concreto a fin de que puedan fabricarse herramientas de desarrollo que permitan el trabajo con los componentes de manera visual.

JavaBeans™

Los JavaBeans son los componentes fabricados con la tecnología Java. El nombre se podría traducir como grano de Java (o grano de café, ya que en Estados Unidos se llama Java al café por el consumo que se hace del café procedente de la isla de Java), es decir es lo que forma una aplicación Java completa.

La idea es la misma que la comentada anteriormente como referencia a los componentes, se colocan directamente JavaBeans en una aplicación y se utilizan sin conocer su interior. Como además se utiliza un modelo concreto, numerosas

aplicaciones de desarrollo permitirán trabajar con los JavaBeans a fin de facilitarnos su manejo.

Dichas herramientas nos permitirán desde un entorno visual (al más puro estilo Delphi o Visual Basic) manipular las propiedades del componente sin tocar en absoluto el código. Al manipular el Bean podremos examinar sus propiedades, eventos y métodos. Esto se consigue mediante una característica fundamental de los JavaBeans que es la reflexión (véase *reflexión,* página 82). La reflexión permite conseguir información del JavaBean aunque no dispongamos de la misma en forma documental (la obtenemos directamente de su código).

Los JavaBeans admiten su **serialización** (véase *serialización*, página 107), es decir se puede almacenar su estado en tiempo de ejecución.

Aunque JavaBeans desde la versión 1.1 de Java apareció con una vocación visual, lo cierto es que se pueden utilizar no solo como componentes visuales herederos de la clase **Component** (aunque sigue siendo parte de su utilización fundamental); así por ejemplo las páginas JSP están muy preparadas para trabajar con JavaBeans.

áreas principales de los JavaBeans

El modelo ideado por Sun para los JavaBeans incluye cinco áreas fundamentales que los JavaBeans han de cumplir

- **Manejo de eventos.** La posibilidad de que los JavaBeans entreguen eventos a otros componentes. Se amplía el modelo de eventos de AWT (véase *eventos*, página 139) para conseguir un modelo más personalizado y sencillo de manipulación de eventos.
- **Propiedades.** Las propiedades definen las características de los componentes. Los JavaBeans implementan un manera normalizada de utilizar las propiedades.
- **Persistencia.** Relacionada con la posibilidad de almacenar el estado del JavaBean (se basa en la serialización).
- **Introspección.** Permite a los propios JavaBeans proporcionar información sobre sus propiedades y métodos. Tiene que ver con la reflexión comentada anteriormente.
- **APIs de diseño de componentes.** Los JavaBeans están pensados para utilizarse en entornos que permiten su personalización. De este modo el diseño de aplicaciones se realiza en tiempo de diseño de manera más efectiva.

empaquetamiento de JavaBeans

Los Java Beans se empaquetan en archivos JAR, como las Applets. En el caso de los Java Beans, el archivo **manifest** se tiene que editar obligatoriamente. En dicho archivo hay que indicar qué clases del paquete JAR son Beans. En el contenido de este archivo, hay que añadir una línea **Name** indicando el nombre de la clase que implementa el Bean (incluyendo el paquete en el que está la clase, la ruta del paquete se indica con símbolos / en lugar del punto, por ejemplo com/miPaquete/clase.class) y una línea con el texto `Java-Bean: True`.

Ejemplo:

```
Manifest-Version 1.0

Name: CuadroTextoFijo.class
Java-Bean: True

Name: CuadroNumérico.class
Java-Bean: True
```

Se supone que ambas clases implementan JavaBeans. En el archivo JAR hay que empaquetar todos los archivos necesarios.

uso del JavaBean en un entorno de desarrollo

Desde hace años Java permite la descarga de una aplicación llamada BDK (*Beans Developer Kit*, kit de desarrollo de Beans). Esta herramienta incluye un programa (llamado **beanbox**) para probar JavaBeans.

Hoy en día no se utiliza mucho ya que los IDE para desarrollar aplicaciones Java como NetBeans o JBuilder incluyen herramientas más avanzadas de manejo de JavaBeans.

En estos entornos, los JavaBeans se arrastran directamente desde las barras de herramientas hacia un contenedor, desde ahí se manipulan. En el caso de NetBeans, el menú ***Tools-Insert New JavaBean*** permite seleccionar un archivo JAR que contenga JavaBeans y luego elegir uno de los JavaBeans para colocarle en la barra de herramientas del editor que se desee.

Una vez insertado se le manipulan las propiedades iniciales desde el propio editor.

propiedades de los JavaBeans

Los JavaBeans no poseen una clase raíz. No obstante, los visuales tienen como padre (directa o indirectamente) la clase Component. La realidad de los JavaBeans es que son **clases normales que pueden ser examinadas de manera especial por las herramientas de desarrollo.** Las propiedades de los JavaBeans pueden ser examinadas por esas herramientas para poder ser editadas de forma visual.

Para que esas herramientas detecten de manera correcta las propiedades, éstas tienen que utilizar el modelo de desarrollo correcto de JavaBeans. En ese modelo se define la forma de acceder a las propiedades.

propiedades simples.

Son propiedades que permiten su uso mediante métodos **get/set** (obtener/cambiar). Las propiedades permiten cambiar su valor usando el método **set** correspondiente (por ejemplo para cambiar la propiedad *título* se utilizaría **setTítulo(**"Imagen principal"**)**). El valor de la propiedad se obtiene mediante método **get** (por ejemplo **getTítulo()** devolvería la cadena *Imagen principal*).

Hay una excepción a lo comentado anteriormente; si la propiedad es booleana, entonces para obtener su valor se usa el método **is** (por ejemplo **isTítuloActivo()**) que lógicamente devuelve **true** o **false** dependiendo del estado de la propiedad.

propiedades indexadas.

Son propiedades que admiten una serie concreta de valores. En este caso hay cuatro métodos de acceso. Si la propiedad es por ejemplo ***Notas*** y representa un array de números **double.** Entonces tendríamos:

- *double[]* ***getNotas()*** Devolvería el array de notas.
- *double* ***getNotas(****int i)* Devuelve la nota número i
- *void* ***setNotas(****int i, double nota)* Cambía el valor de la nota número i para que tome el valor indicadp
- *void* ***setNotas(****double[] notas****)*** Hace que el contenido de la propiedad sea el contenido del array utilizado.

Propiedades dependientes.

Permiten mandar mensajes de modificación de propiedades a otros Beans cuando una propiedad cambia, de este modo se ligan dos componentes; cuando el primero cambia, el segundo se da cuenta y realiza una determinada acción

Por ejemplo si tenemos un Java que muestra imágenes y un componente que sirve para rellenar rutas de archivo, podemos hacer que al cambiar la propiedad de la ruta, el cuadro de la imagen muestre la imagen correspondiente a esa ruta. El proceso para realizar este mecanismo es:

1> Cuando cambia el valor de la propiedad, el componente envía un evento de tipo **PropertyChangeEvent** a todos los objetos registrados como escuchadores (*listeners*) del componente. Las clases relacionadas con estos eventos están en el paquete **java.beans**

2> Para permitir que haya componentes que escuchen al JavaBean hay que definir dos métodos:

- void **addPropertyChangeListener(**PropertyChangeListener escuchador)
- void **removePropertyChangeListener(**PropertyChangeListener escuchador)

El primer método sirve para añadir un escuchador. El segundo sirve para quitar un escuchador.

3> El componente *escuchador* debe implementar la interfaz **PropertyChangeListener** que le obligará a definir el método **propertyChanged,** método que gestiona este tipo de eventos.

Definir el funcionamiento de esos métodos es una tarea muy compleja, por eso existe una clase llamada java.beans.**PropertyChangeSupport** que tiene definido el código de esos métodos a fin de que cualquier clase lo pueda aprovechar. Para ello en el Bean necesitamos un objeto de este tipo, definida de la siguiente forma:

```
private PropertyChangeSupport cambios=
        new PropertyChangeSupport(this);
```

En el constructor se indica el objeto al que se le quiere dar soporte (lo normal es **this**). Y ahora la definición de los métodos anteriores, la definición sería:

```
public void addPropertyChangeListener(
                                PropertyChangeListener pcl){
  cambios.addPropertyChangeListener(pcl);
}
public void removePropertyChangeListener(
                                PropertyChangeListener pcl){
  cambios.removePropertyChangeListener(pcl);
}
```

Para lanzar eventos **PropertyChangeEvent** a los escuchadores, se utiliza el método **firePropertyChange** cada vez que se cambia el valor de la propiedad. A este método se le pasan tres parámetros: el nombre de la propiedad, el valor antiguo y el valor nuevo. Los valores antiguos y nuevos deben de ser objetos, lo que obliga a utilizar las clases envoltorio **Integer, Double, Boolean, Char,** etc. para poder pasar valores de tipos básicos.

Un detalle importante a tener en cuenta es que si la clase deriva de la clase Swing **JComponent** (página 129), lo cual es muy común ya que casi todos los JavaBeans heredan de esa clase, entonces no es necesario implementar los métodos **addPropertyChangeListener** y **removePropertyChangeListener** ya que ya están implementados en la clase padre. En este caso, para notificar un cambio de propiedad basta usar el método **firePropertyChange** de la clase **JComponent** (además en el método **firePropertyChange** de *JComponent* sí que se pueden pasar como valores los tipos básicos **int, double, boolean, char, ...**).

Lógicamente cualquier JavaBean que desee escuchar eventos de tipo **PropertyChangeEvent** deben implementar la interfaz **PropertyChangeListener.** Esa interfaz obliga a definir el método:

- **void propertyChange(PropertyChangeEvent e)**

Este método es llamado cuando se modifica el valor de la propiedad.

Por su parte el evento **PropertyChangeEvent** que recibe este método nos proporciona estos métodos:

- **Object getOldValue()** Obtiene el valor antiguo de la propiedad (el valor anterior al cambio)
- **Object getNewValue()** Obtiene el nuevo valor de la propiedad.
- **String getPropertyName()** Obtiene el nombre de la propiedad que cambió (puede ser **null** si cambiaron varias propiedades a la vez).

propiedades restringidas

Es la propiedad más compleja de implementar, pero admite posibilidades muy interesantes. La idea es similar a la de las propiedades dependientes; se valoran cambios de valor en un JavaBean. Sin embargo, en este caso se basa en que el objeto oyente de eventos (eventos de *veto*) puede **vetar** los cambios si no se ajustan a unas

determinadas características. Cuando una propiedad puede dar lugar a un veto, es cuando se dice que la propiedad es restringida.

La mayor parte de restricciones en una propiedad se pueden programar desde el propio JavaBean. Esta idea es muy interesante cuando la restricción depende de los valores de varios componentes.

Para que un componente cualquier pueda indicar que hay objetos escuchadores, necesita tener como métodos miembro, los siguientes:

- **void addVetoableChangeListener(**VetoableChangeListener escuchador**).** Hace que el objeto escuchador, sea capaz capturar eventos de tipo **PropertyVetoEvent**

- **void removeVetoableChangeListener(**VetoableChangeListener escuch**)** Hace que el objeto *escuch* deje de escuchar los eventos de veto.

Una vez más, como definir estos métodos es muy complejo, se puede crear un objeto **VetoableChangeSupport** para conseguir un soporte sencillo a fin de definir los métodos anteriores:

```
//en el constructor del JavaBean
private VetoableChangeSupport soporteVeto=
        new VetoableChangeSupport(this);

//...fuera del constructor
public void addVetoableChangeListener(
                                VetoableChangeListener pcl){
  cambios.addVetoableListener(pcl);
}
public void removeVetoableChangeListener (
                                VetoableChangeListener pcl){
  cambios.removeVetoableChangeListener (pcl);
}
```

Los componentes que deriven de JComponent no necesitan este código ya que todos estos métodos están definidos.

El lanzamiento del evento se realiza mediante el método **fireVetoableChange** (de la clase JComponent o de la clase VetoableChangeSupport), que requiere tres parámetros: el nombre de la propiedad que cambia (en forma de String), el valor antiguo de la propiedad y el valor nuevo de la propiedad. Los valores antiguo y nuevo deben ser de tipo **Object** lo que significa que si la propiedad es de tipo básica habrá que usar un clase evolvente (**Integer, Boolean**,...). *fireVetoableChange* debe ser llamado cada vez que se cambia la propiedad.

La escucha de este tipo de eventos la realiza cualquier objeto que pertenezca a clases que implementen la interfaz **VetoableChangeListener;** esta interfaz obliga a definir el método siguiente:

- **public void vetoableChange(**PropertyChangeEvent ev**) throws PropertyVetoException**

Se puede observar que este método recibe un objeto del tipo **PropertyChangeEvent** (no hay eventos *VetoableChangeEvent*) y que lanza excepciones del tipo **PropertyVetoException.** De hecho la clave del funcionamiento de este tipo de propiedades la tiene esta excepción. El método **vetoableChange** tiene que lanzar (mediante la instrucción **throw**) excepciones de ese tipo cuando el cambio de valor en la propiedad no se pueda realizar debido a que no cumple una condición.

El lanzamiento de estas excepciones tiene esta sintaxis:

- **throw new PropertyVetoException(**String mensaje**,** PropertyChangeEvent evento**)**

El objeto de excepción lanzado permite obtener el mensaje (método **getMessage**) y el evento que desencadenó la excepción (**getPropertyChangeEvent**).

En resumen, los pasos para la realización de una propiedad restringida serían los siguientes:

1> El JavaBean que posee la propiedad restringida debe tener definidos los métodos **addVetoableChangeListener** y **removeVetoableChange-Listener.** Si no los tiene definidos (por no ser clase heredera de JComponent), se puede ayudar de un objeto **VetoableChangeSupport** para definirlos.

2> El método set*Propiedad* de la propiedad restringida, debe lanzar el evento de veto mediante **fireVetoableChange**. A la que se pasa un texto con la propidad que cambia, el valor antiguo y el valor nuevo. Ese método debe capturar o lanzar excepciones **PropertyVetoException** ya que la instrucción anterior puede dar lugar a esa excepción:

```
public void setValor(String nuevo){
  try{
  String valorAntiguo=getValor();
  fireVetoableChange("valor",nuevo,valorAntiguo);//posible
                                                 //excepción
  ....//Si el código siguiente se ejecuta no hubo excepción
  }
  catch(PropertyVetoException pve){
  ...//Código que se ejecuta en caso de excepción
  }
```

3> El objeto oyente de los eventos de tipo *veto*, debe implementar la interfaz **VetoableChangeListener.** Esta interfaz obliga a definir el método **vetoableChange** que será llamado cada vez que se lance el evento de veto (que ocurrirá normalmente cada vez que la propiedad cambie):

```
public void vetoableChange(PropertyChangeEvent evt) throws
PropertyVetoException{
   //.. Comprobación de las condiciones
   throw new PropertyVetoException("mensaje",evt);//Sólo si
                    //las condiciones de veto se cumplen
```