Java2

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

Autor: Jorge Sánchez (www.jorgesanchez.net) año 2003

Basado en el lenguaje Java (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.

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 problema.

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. De hecho a partir del C tradicional se creó el C++. 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).
- Tenía las mismas ventajas que el C.

C++ pasó a ser el lenguaje de programación más popular a principios de los 90. Pero la popularidad de Internet iba a propiciar un profundo cambio.

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.

El problema fundamental del C++ era que al compilar se producía un fichero ejecutable cuyo código sólo vale para la plataforma creada. 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 dispositivos aparecido, hay que rescribir el código.

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

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.

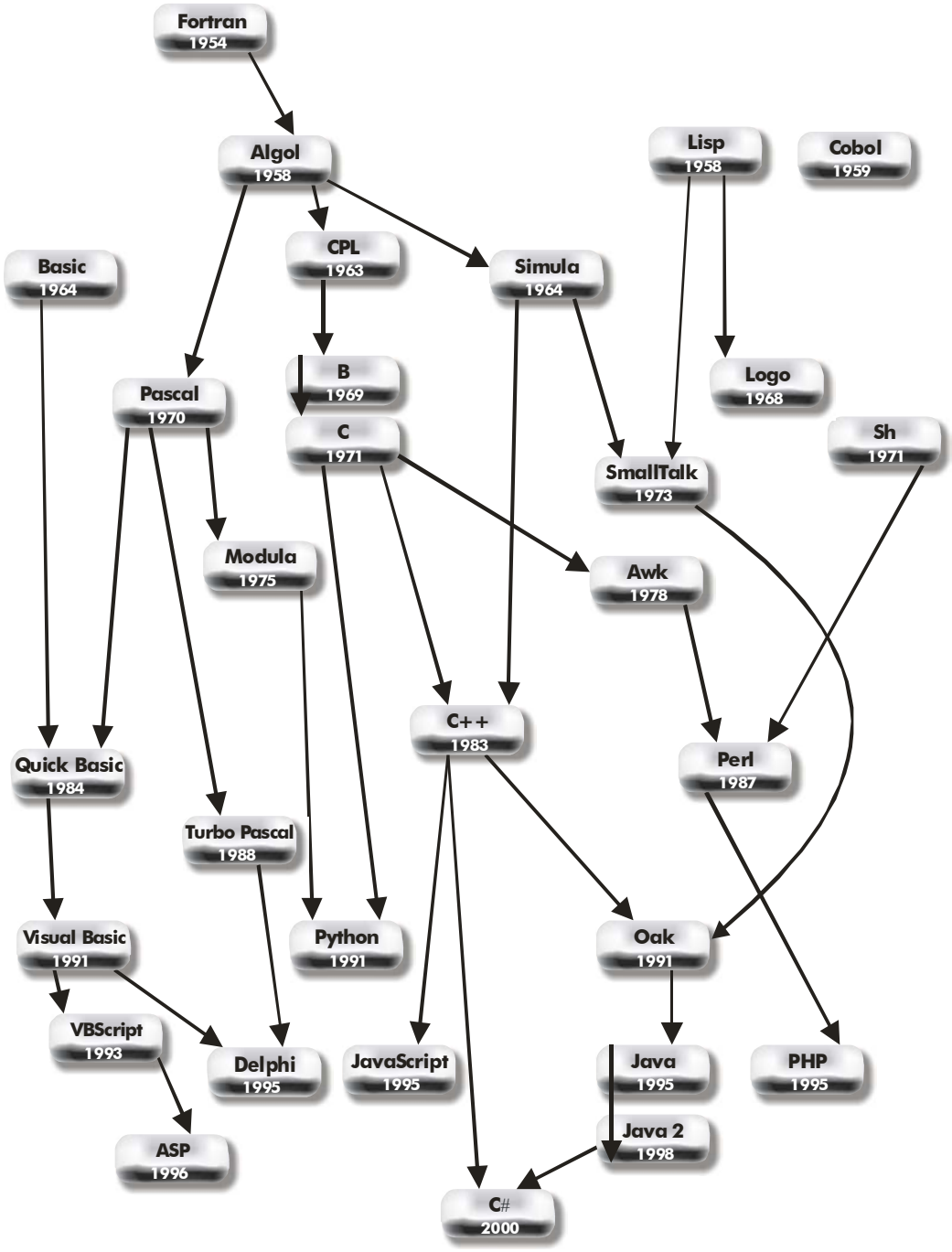


Ilustración 1, ***Evolución de algunos lenguajes de programación***

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 librearías de la plataforma

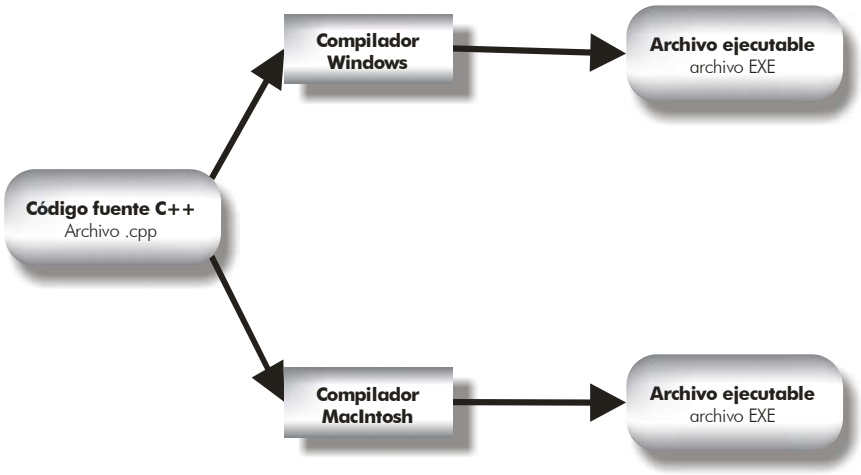


Ilustración 2, *Proceso de compilación de un programa C++*

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 compilar 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 generando el código ejecutable de la aplicación

La máquina virtual de Java, además es un programa muy pequeño y que se distribuye gratuitamente para prácticamente todos los sistemas operativos.

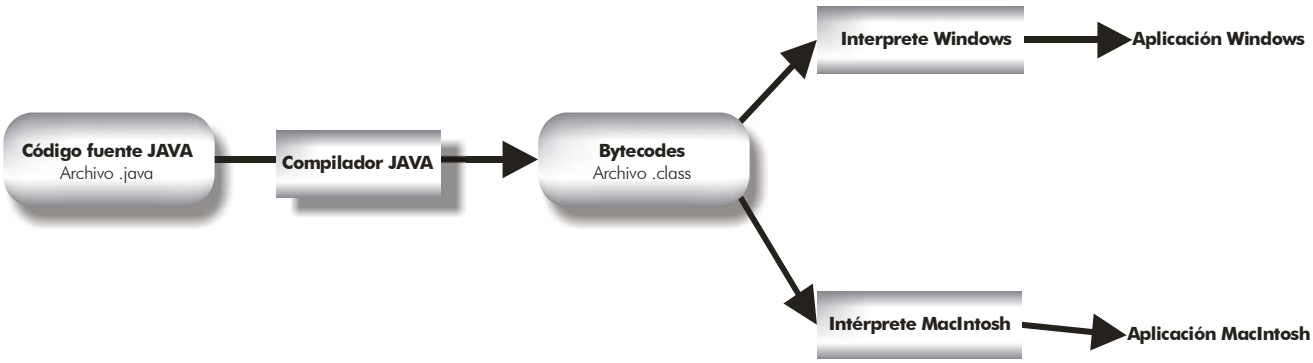


Ilustración 3, *Proceso de compilación de un programa Java*

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.

A este método de ejecución de programas en tiempo real se le llama *Just in Time* (JIT).

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).

ventajas de Java

- Es muy similar a los lenguajes C y C++, con las ventajas que tenían estos y con la virtud de facilitar su aprendizaje a todos los programadores de C (que son muchos).
- Eliminan los punteros, lo que aumenta su seguridad
- Es **totalmente** orientado a objetos. Desde el principio hay que seguir un método POO.
- Está especialmente preparado para crear aplicaciones TCP/IP
- Implementa de forma nativa *excepciones*
- Es interpretado, es un pequeño programa el que interpreta Java y no el sistema operativo.
- Control de tipos de datos más riguroso que en C
- Permite el uso de firmas digitales para asegurar la autoría.
- Multihilo, es decir permite realizar varias tareas a la vez al ordenador.
- Es dinámico, los objetos se cargan en memoria cuando son necesarios.

empezar a trabajar con Java

el kit de desarrollo Java (JDK)

Para escribir en Java hace falta un entorno que permita la creación de los bytecodes y también con capacidades de mostrar aplicaciones de consola y gráficas. 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.

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 1.3

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,...). 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 "*/"

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

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

- A veces se marcan bloques de código, los cuales comienza con { y terminan con }

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

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 es un código de consola se puede probar 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. En esos comentarios se pueden utilizar códigos HTML. 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
  public static void main(String args[]){
        System.out.println("¡Mi segundo programa! ");
  }
}
```

El resultado de ese código tras usar javadoc es la página web:

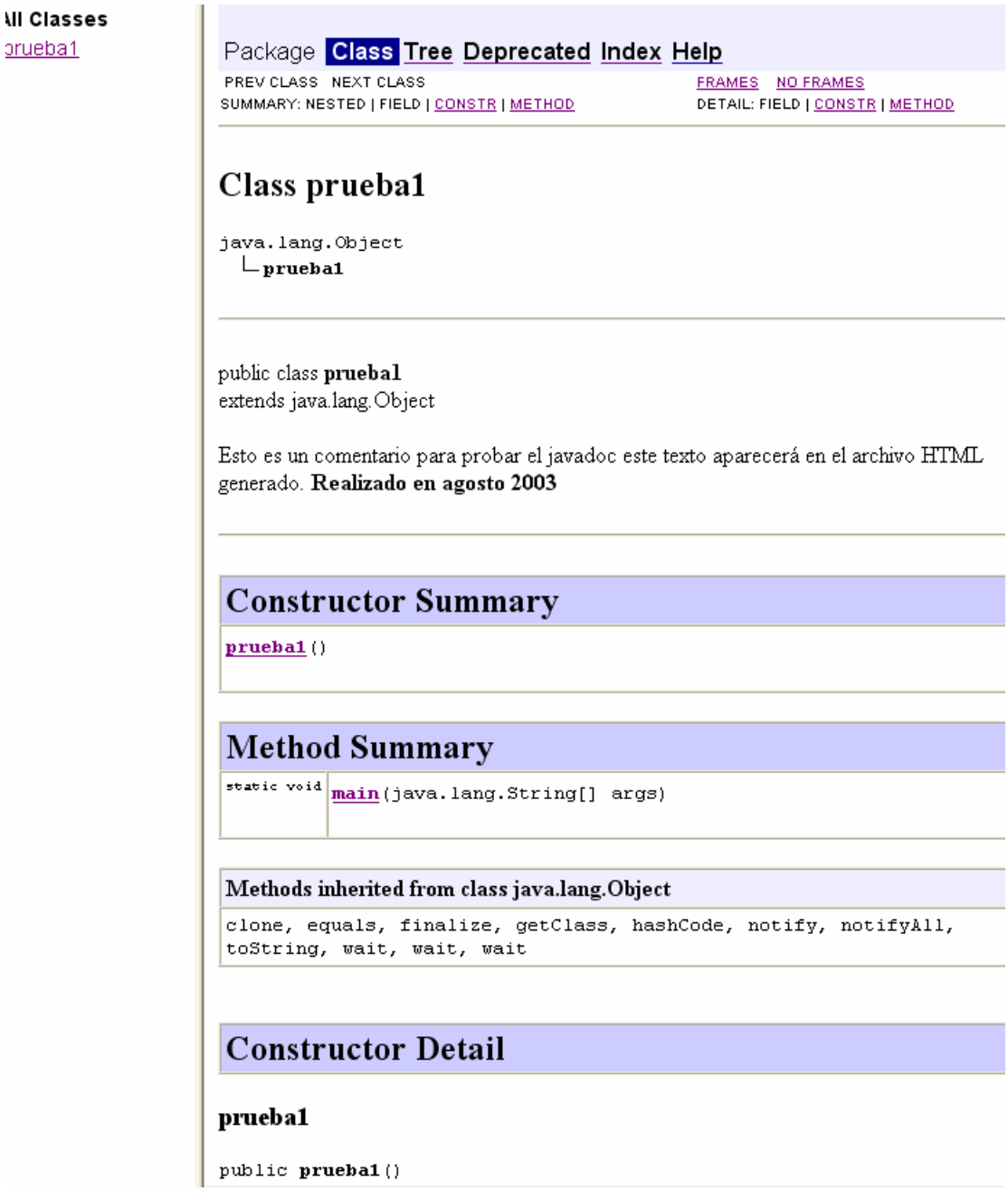


Ilustración 4, Página de documentación de un pro grama Java

importar paquetes y clases

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 que sólo puede contener letras, números y el carácter de subrayado. Además su nombre debe empezar por una letra.

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 entero salvo si al final se le coloca la letra L.

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)
```

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.

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
```

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.

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
!=	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

operador	significado
>>>	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?instruccionSiVerdadera:instruccionSiFalsa;
```

Ejemplo:

```
(dia==Sabado) ||
(dia=Domingo)?laborable=false:laborable=true;
```

instanceof

Se usa para determinar el tipo de un objeto durante la ejecución del programa. Devuelve **true** en el caso de que el objeto posea el tipo indicado. Ejemplo:

```
boolean b=(numero instanceof int);
```

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

operador	significado
tiponúmero **abs(** ***tiponúmero*** *x***)**	Devuelve el valor absoluto de *x*.
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

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==2) dia="Sábado";
else if (diasemana==2) 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**), si toma el primera valor ejecuta las sentencias correspondientes al **case** de ese valor. Y así con las demás excepto con el grupo **default** que se ejecuta si la expresión no tomó ningún valor de la lista.

La sentencia opcional **break** se ejecuta para hacer que el flujo del programa salte al final de la sentencia **switch,** de otro modo se ejecutarían todas las sentencias restantes, sean o no correspondientes al valor que tomo la expresión.

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. En el caso de que la condición pase a ser falsa, el bucle deja de ejecutarse. 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 también un bucle, sólo 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. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Un contador determina las veces que se ejecuta el bucle. Sintaxis:

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

La expresión inicial es una instrucción que se ejecuta una sola vez, al entrar en el **for** (normalmente esa expresión lo que hace es dar valor inicial al contador del bucle). La condición es una expresión que devuelve un valor lógico. En el caso de que esa expresión sea verdadera se ejecutan las instrucciones.

Después de ejecutarse las instrucciones interiores del bucle, se realiza la expresión que tiene lugar en cada vuelta (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. Por eso hay que utilizar el operador **new**. Con él se indica ya el tamaño y 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];
```

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};
```

Esto declara e inicializa un array de tres elementos.

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];
```

En el ejemplo anterior cuando se vuelve a definir notas, hay que tener en cuenta que se pierde al valor que tuviera el array anteriormente.

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.

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 debe usar alguna de estas expresiones:

- *s1*.**equals(***s2***).** Da **true** si *s1* es igual a *s2*
- *s1*.**equalsIgnoreCase(***s2***).** Da **true** si *s1* es igual a *s2* (ignorando mayúsculas y minúsculas)
- *s1*.**compareTo(***s2***).** Si *s1* < *s2* devuelve un número menor que 0, si *s1* es igual a *s2* devuelve cero y si *s1*>*s2* devuelve un número mayor que 0
- *s1*.**compareToIgnoreCase(***s2***).** Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

métodos de String

Los objetos de tipo String pueden utilizarse con métodos que permiten mayor potencia de uso. Veremos los métodos más importantes:

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).

length

Permite devolver la longitud de una 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). 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). 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. 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 que se indica.

replace

Cambia todas las apariciones de un carácter por otro en el texto que se indique:

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

toUpperCase

Devuelve la versión mayúsculas de la cadena.

toLowerCase

Devuelve la versión minúsculas de la cadena.

lista completa de métodos de String

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
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

método	descripción
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**

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*
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

método	descripción
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**

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.

clases

Las clases son las plantillas para hacer objetos. En una clase se define los comportamientos y propiedades que poseerán los objetos. 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 propiedades.** 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).

El formato general para crear una clase es:

```
[acceso] class nombreDeClase {
  [access] [static] tipo propiedad1;
  [access] [static] tipo propiedad2;
  [access] [static] tipo propiedad3;
  ...
  [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 **propiedades 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(){...
```

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)

Para poder acceder a las propiedades de un objeto, se utiliza esta sintaxis:

```
objeto.propiedad
```

Por ejemplo:

```
Noria.radio;
```

métodos

Los métodos se utilizan de la misma forma que las propiedades:

```
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:

```
Noria.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.

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 propiedades de la clase (variables 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;
```

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.

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();
    }
}
```

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).

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.

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
```

métodos y propiedades genéricos (*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. Así funciona la clase **Math** (véase la clase Math, página 20). 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.

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.

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 una recolección de basura (***garbage***=basura) la que se encarga de gestionar los objetos que se dejan de usar. 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 eso no es lo mismo que el famoso **delete** del lenguaje C++; no se libera inmediatamente la memoria, sino que pasará un cierto tiempo (impredecible, por otro lado). Se puede invocar al recolector de basura 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 {
   public static void main(Stgring[] args)  {
         uno prueba = new uno();//referencia circular
         prueba = null; //no se liberará bien la memoria
   }
```

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

```
}
```

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.**

reutilización de clases

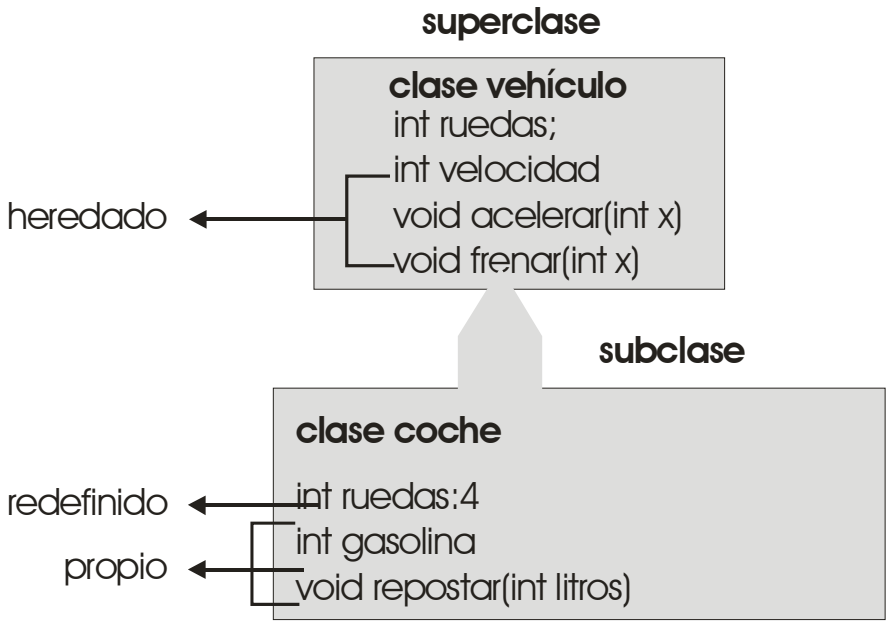
herencia

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
```



métodos y propiedades no heredados

Por defecto se heredan todos los métodos y propiedades *friendly, 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.

Ilustración 5, herencia

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:

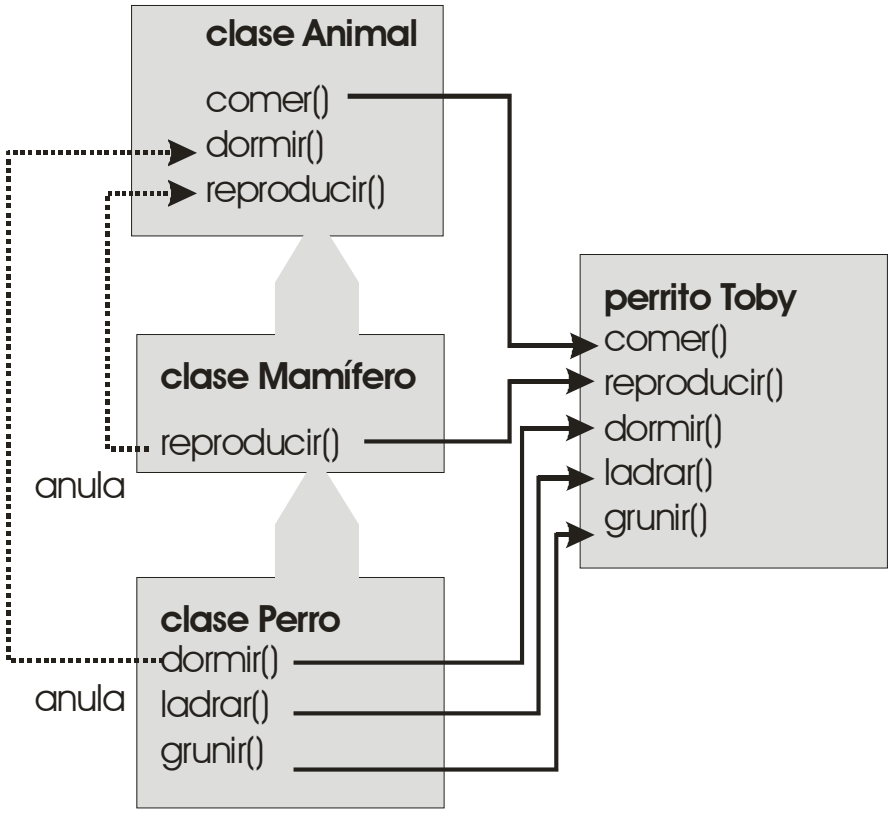


Ilustración 6, anulación de métodos

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 15, 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**

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.

relaciones entre clases

relaciones es-a y relaciones tiene-a

Se trata de una diferencia sutil, pero hay dos maneras de que una subclase herede de otra clase.

Siempre que aquí se ha hablado de herencia, en realidad nos referíamos a relaciones del tipo es-a. Veamos un ejemplo:

```
class uno{
  public void escribe() {
        System.out.println("la clase uno, escribe");
  }
}
class dos extends uno{
  dos() {
        escribe();
  }
}

public class prueba{
  public static void main(String[] args){
        dos d= new dos();//El método escribe escribirá
  }
```

La clase *dos* tiene una relación es-a con la clase *uno* ya que es una subclase de uno. Utiliza el método escribe por herencia directa (relación es-a).

Este mismo ejemplo en la forma de una relación tiene-a sería:

```
class uno{
  public void escribe() {
        System.out.println("la clase uno, escribe");
  }
}
class dos {
  uno u;
  dos() {
        u = new uno();
        u.escribe();
  }
}
```

```
public class prueba{
  public static void main(String[] args){
        dos d= new dos();//El método escribe escribirá
  }
}
```

El resultado es el mismo, pero el método *escribe* se ha utilizado, no por herencia, sino por el hecho de que la clase dos **tiene** un objeto de clase *uno*.

Se habla de relaciones **tiene-a** cuando una clase está contenida en otra.

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.

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 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 llamado por ejemplo **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 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 puede anular los métodos abstractos de éste, redefiniéndolos en la propia clase.

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. Las variables se tienen obligatoriamente que inicializar. Ejemplo:

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

Los métodos son simples prototipos y la variable se considera una constante (a no ser que se redefina en una clase que implemente esta interfaz)

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.

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 esto es que pueden asignarse variables interfaz a cualquier objeto que tenga en su clase implementada 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 tengan implementados los métodos del interfaz de la variable.

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. Mediante el comando **import** (visto anteriormente), se permite utilizar una determinada clase en un programa. Esta sentencia se coloca arriba del código de la clase.

```
import ejemplos.tema5.vehiculo;
import ejemplos.tema8.* //Usa todas las clase del paquete  tema8
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta 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.** Para que sean visible para cualquier clase de cualquier paquete, deben declararse con **public.**

organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello a veces 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).

Así para el paquete **prueba.reloj** tiene que haber una carpeta prueba, dentro de la cual habrá una carpeta reloj.

Una clase se declara perteneciente aun determinado paquete usando la instrucción **package** al principio del código:

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

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 inhabitual, 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**

En Java, cuando esto ocurre, el código es *lanzado* 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.

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 de tipo **Exception**. 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.

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 (TipoExcepción 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) {
...
}
```

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. Dentro del bloque try puede haber varias sentencias que provoquen el mismo tipo de excepción pero bastará una sola sentencia catch para manejarlas.

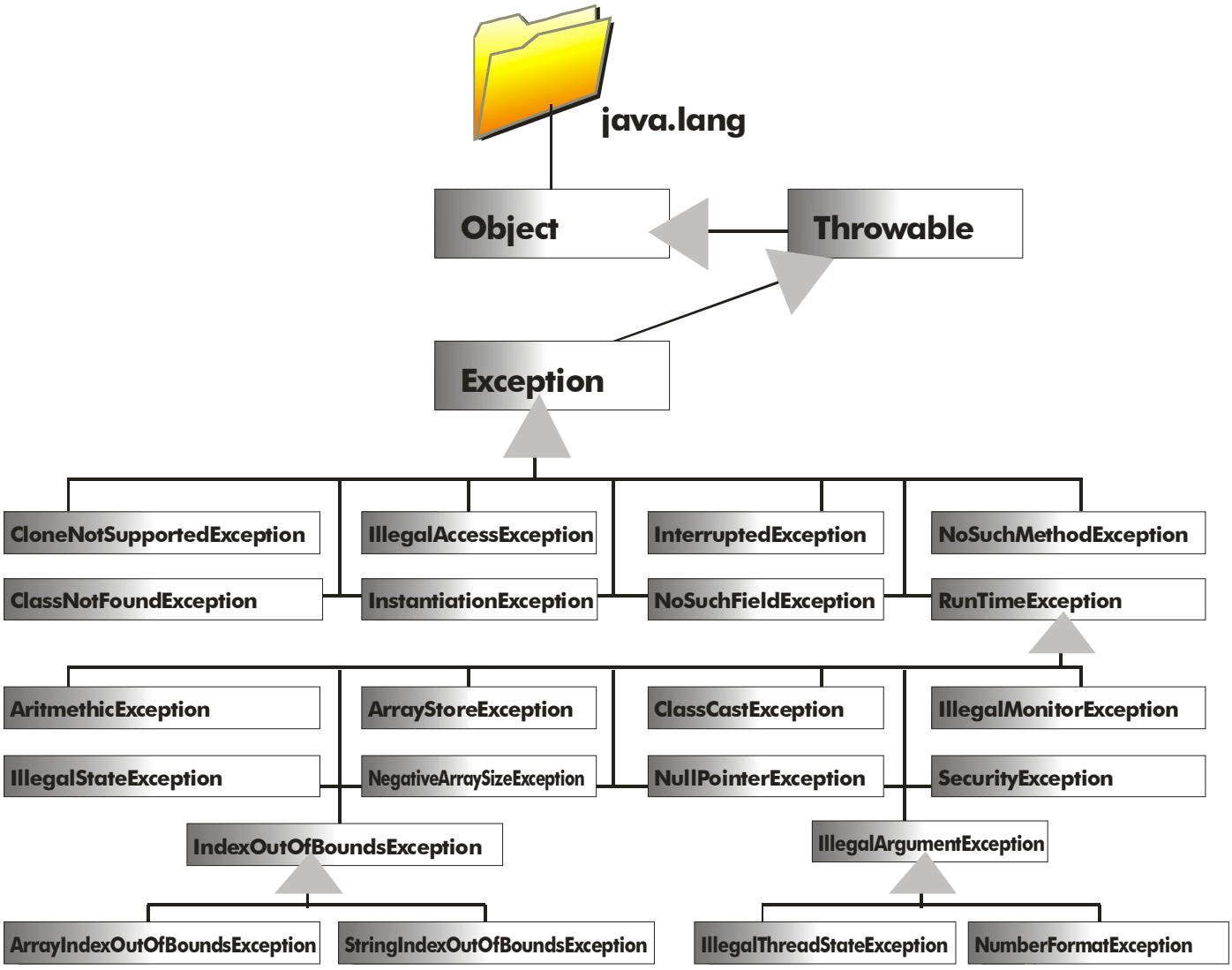


Ilustración 7, Jerarquía de las clases de manejo de excepciones

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 Ilustración 7, 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

Hay código al que hay que marcarle la posibilidad de que pueda provocar un determinado tipo de excepción. Si no, las excepciones, en ese código, no se pueden comprobar. Esto se hace añadiendo la palabra **throws** tras la primera línea de un método. 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.

throw

Esta instrucción nos permite 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:

```
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.

clases fundamentales

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 en muchas ocasiones hay que redefinirlos para que funcionen adecuadamente

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 (esto habrá que haberlo indicado redefiniendo el método equals en la clase Coche). El operador "==" devuelve **true** si los dos objetos se refieren a la misma cosa (las dos referencias apuntan al mismo objeto).

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 ((o.matricula==matricula) &&
                    (o.marca==marca) && (o.modelo==modelo))
                    return true
        }
        return false; //Si no se cumple todo lo anterior
```

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.**

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.

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.

hashcode

Un *hashcode* es un número de comprobación de un objeto. Es un firma numérica de éste. Es distinto código para objetos distintos, pero debería ser el mismo para objetos con los mismos datos (objetos que al comprobarse con *equals* devuelven *true*).

Los *hashcodes* se utilizan en las tablas hash que son tablas dinámicas que permiten almacenar datos y asociarles un código.

lista 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. 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.

método	significado
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.
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

método	significado
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**
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.

método	significado
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.

método	significado
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**

clase	representa al tipo básico..
java.lang.Float	**float**
java.lang.Double	**double**

Hay que tener en cuenta que no son equivalentes a los tipos básicos. Su construcción se basa en objetos, es decir tienen constructor:

```
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 **valueOf** entre otros para convertir un String en uno de esos tipos. Este método es estático:

```
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** seguido del tipo básico a convertir. Por 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).

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);//Simlla obtenida de un long
```

métodos de Random

método	devuelve
nextBoolean()	**true** o **false** aleatoriamente
nextInt()	un **int**
nextInt(int n**)**	Un número entero de 0 a *n* (incluído)
nextLong()	Un **long**
nextFloat()	Número decimal de -1,0 a 1.0
nextDouble()	Número doble de -1,0 a 1.0
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 que define la funcionalidad de las fechas de calendario y define una serie de campos muy interesantes para trabajar con las fechas. Entre ellas:

- **Día de la semana:** DAY_OF_WEEK número del día de la semana (del 0 al 6). Se pueden usar las constantes MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
- **Mes:** MONTH es el mes del año (el primer, JANUARY, es el 0). 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 de la semana
- **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 de la semana
- **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 MILISECOND.

Esta clase también define una serie de métodos abstractos y estáticos. Entre los que destacan **get** y **set**.

clase GregorianCalendar

Es subclase de la anterior. 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 julio de 2003
GregorianCalendar fecha2=new
GregorianCalendar(2003,Calendar.JULY,2);
  //Igual que la anterior
GregorianCalendar fecha2=new
GregorianCalendar(2003,7,2,12,30);
  //2 de Julio de 2003 a las 12:30
GregorianCalendar fecha2=new
GregorianCalendar(2003,7,2,12,30,15);
  //2 de Julio 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 campo 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 u opcionalmente hora y minutos.

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

método getTime

Devuelve un objeto Date que es el equivalente a la fecha de calendario.

método setTime

Hace que el objeto de calendario tome como fecha la representada por un objeto date.

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

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.**

clase DateFormat

Clase que implementa métodos sofisticados para formatear fechas. Tiene una gran cantidad de opciones. Por defecto un objeto DateFormat con opciones básicas se crea con:

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

El método estático **getInstance()** permite crear objetos de fecha con formato de fecha corta y opciones básicas.

Otras opciones consisten crear formatos de fecha con **getDateInstance**, **getTimeInstance** o **getDateTimeInstance**. Son todos métodos estáticos que recibe como parámetro el nivel de detalle de la fecha y se suele indicar con las constantes SHORT, MEDIUM, LONG y FULL.

```
DateFormat df=DateFormat.getDateInstance(DateFormat.LONG);
System.out.println(df.format(new Date()));
//16 de Septiembre de 2003
```

La fecha sale con el formato por defecto del sistema. Se puede añadir un segundo parámetro de tipo **Locale** que hace que el formato de fecha se cree con las propiedades de las fechas de un determinado país. Ejemplo:

```
DateFormat df=DateFormat.getDateInstance(DateFormat.LONG,
Locale.ITALY)
System.out.println(df.format(new Date()));
// 16 settembre 2003
```

El método **format** es el encargado de convertir el objeto en forma de String. Este método recibe un objeto **Date** como parámetro.

El método contrario a **format** es **parse** que convierte un String en forma de fecha. Si el String no es válido ocurrirá una excepción del tipo **ParseException**. También pueden ocurrir excepciones **NullPointerException** y **StringIndexOutOfBounds**.

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.

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 están en el paquete **java.io**

InputStream/ OutputStream

Clases abstractas que definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). 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()** y **write().**

Reader/Writer

Clases abstractas que definen las funciones básicas de escritura y lectura basada en 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.

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 un InputStream a forma de Reader.

DataInputStream/DataOutputStream

Filtros de secuencia que permiten leer y escribir tipos simples (**int, short, byte,..., String)**.

ObjectInputStream/ObjectOutputStream

Filtros de secuencia que permiten leer y escribir objetos

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.

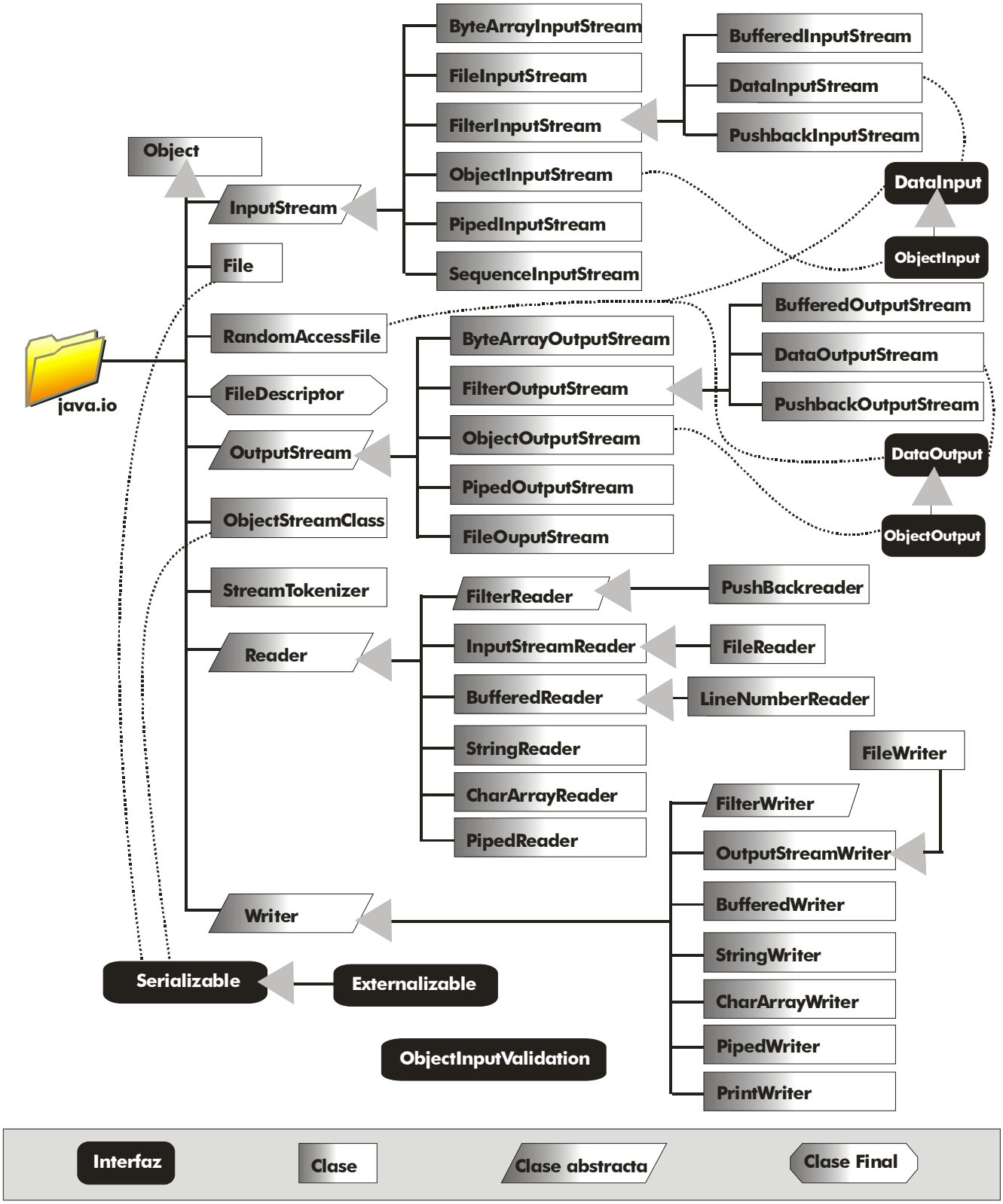


Ilustración 8, Clases e interfaces del paquete java.io

PrintWriter

Secuencia pensada para impresión de texto.

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

java.lang.System es un paquete que contienen clases relacionadas con el propio 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. 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];
```

InputStreamReader y OutputStreamWriter

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 único que hemos creado de esa forma es un objeto *convertidor*. Para leer cadenas de carácter se suele utilizar la clase **BufferedReader**. En esta clase, el método **ReadLine()** permite leer una línea de texto y convertirla a String que es más fácil de manipular. Esta clase usa un constructor que acepta objetos **Reader** (y por lo tanto **InputStreamReader**) 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 habrá que capturar dicha lectura.

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

secuencias con FilterInputStream

La clase FilterInputStream permite cambiar el comportamiento de la clase InputStream, con la finalidad de hacerla más útil para leer tipos diversos de datos. Posee tres clases descendientes:

- **DataInputStream.** Pensada para leer tipos básicos.
- **BufferedInputStream.** Para leer datos usando un búfer de lectura.
- **PushbackInputStream.** Mantiene un búfer de tipo *back* para poder leer los últimos caracteres.

DataInputStream

DataInputStream, se utiliza para leer datos primitivos. Se crean objetos de esta clase usando un constructor que recibe como argumento un objeto de tipo *InputStream*. Sus métodos son:

- **read().** Heredado de *InputStream*
- **readDouble().** Lee un número double.
- **readBoolean().** Lee un valor booleano
- **readInt()**
- **readFloat()**
- **readByte()**
- **readChar()**
- **skipBytes(int** n**).** Salta un número de bytes del archivo.

DataOutputStream

Es el equivalente de la clase anterior, pero en forma de escritura. Usa los mismos constructores (salvo que utiliza un OutputStream en lugar de un InputStream) y utiliza los mismos métodos (usando **write** donde antes era **read**),

```
DataInputStream dato=new DataInputStream(System.in);
DataOutputStream dato2=new DataOutputStream(System.out);
double d=0;
try{
   d=dato.readDouble();
   dato2.writeDouble(d);
}
catch(IOException e){
   System.out.println("error ");
}
```

No obstante no es recomendable usar la entra y salida estándar para estos métodos, están más pensados para usar archivos o comunicación entre redes.

PrintWriter y PrintStream

La clase PrintWriter es subclase de *Writer* y posee métodos muy adecuados de escritura. De hecho el miembro **out** de la clase **System** es un objeto *PrintStream* que es la versión byte de *Print*Writer. Los métodos de estas clases no lanzan excepciones *IOException* lo que facilita la escritura. Entre ellos están **print** y **println**.

Ambos permiten utilizar cualquier tipo de datos convertible a String. La diferencia entre ellos están en que println añade un Intro tras la escritura.

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");//o 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*).

el problema de las rutas

Cuando se crean programas en Java hay que tener muy presente que no sabremos muchas veces 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.

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** para especificar rutas ya que son independientes de la plataforma.

Si no se desea usar Swing, entonces hay que utilizar las variables estáticas que posee File. Estas son:

propiedad	uso
char separator	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es "/" y en Windows es "\", que se debe escribir como \\, ya que ese carácter en Java es para poder escribir caracteres de control.
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 getAbsolutePath()	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.
URL toURL() throws MalformedURLException	Convierte el archivo a su notación URL correspondiente

método	uso
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.
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.
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* y *FileOutputStream*.

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");
```

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. Ejemplo de escritura con *DataOutputStream*:

```
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");
}
```

El listado anterior escribe 233 números aleatorios de tipo **double** en un archivo. Para leer esos números. :

```
try{
    FileInputStream fis=new FileInputStream(f);
```

```
    DataInputStream dis=new DataInputStream(fis);
    boolean finArchivo=false;
    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.** Son

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
- **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.

Swing

introducción

Swing es un conjunto de clases desarrolladas por primera vez para Java 1.2 (el llamado Java2), para reemplazar al anterior paquete que implementaba clases para fabricar interfaces de usuario, el llamado AWT (*Abstract Window Tools*) que aún se usa bastante.

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.

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. Son 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).

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.

Sin embargo AWT tenía varios problemas y por ello 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 son:

- 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.

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.awtr.container** y ésta de **java.awt.component.** De esto se deduce que Swing es una extensión de AWT, de hecho su estructura es análoga.

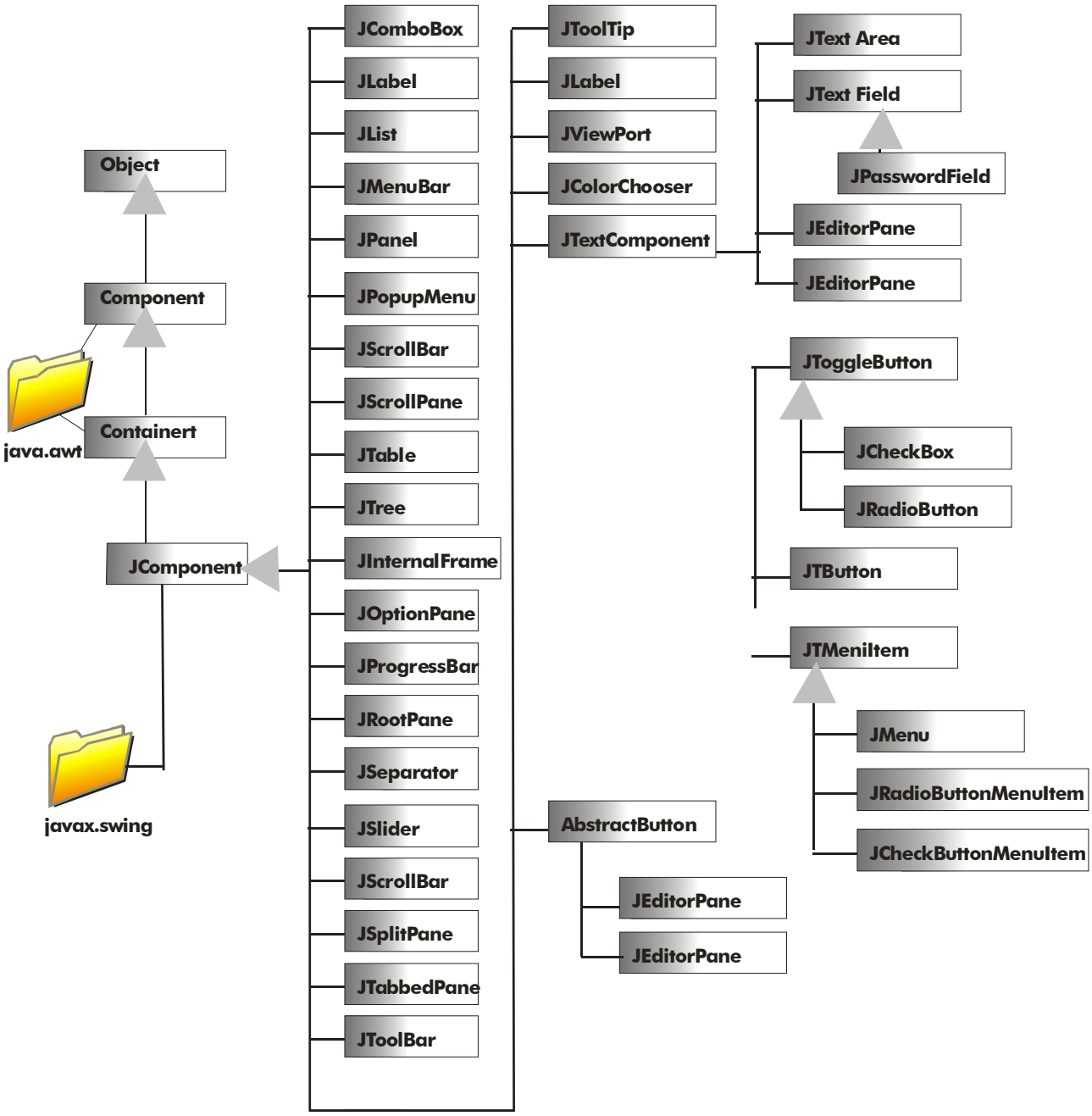


Ilustración 9, Referencia de las clases Swing

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 ***par***. Esto significaba que cada componente creado con AWT, creaba un para 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 el dato en concreto al que se refiere el objeto.
- **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 lo que permitiría capturar el clic del ratón que permite el cambio de selección en ese control.

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 Location
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 setPreferredSize(*Dimension* d**)**	Cambia el tamaño preferido del componente. Este tamaño es el que el componente realmente quiere tener.
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.

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.
- **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**).**

Los objetos Color poseen métodos muy 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.

métodos de dibujo

método	uso
void paint(*Graphics* p)	Pinta el componente y sus subcomponentes. Delega sus funciones en los tres métodos siguientes
void paintComponent(*Graphics* p)	Pinta sólo este componente
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 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
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, ..
- **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

método	uso
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.

método	uso
JFrame(String titulo, ***GraphicsConfiguration*** gc**)**	Crea una ventana con el título y la configuración gráfica indicada.

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
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 enviado a otro objeto para su manejo. Un evento se lanza (o se dispara) cuando ocurre una determinada situación,. Cuando es lanzado, un gestor de eventos trata su manipulación.

Los objetos de eventos derivan de la clase **java.util.EventObject.** Esta clase sólo contiene como información útil, una referencia al objeto que generó el evento. Esta información la da el método **getSource()** que devuelve un objeto **Object**. la clase **AWTEvent** es la clase AWT (que desciende de *EventObject*) que trabaja para los eventos en AWT. Otros eventos muy utilizados son:

- **ActionEvent.** Acción tomada por el usuario sobre un componente.
- **MouseEvent.** El ratón actúa sobre un componente.

gestores de eventos

Los eventos se envían pasándoles como parámetros al método controlador de eventos. Así cada tipo de evento tiene asociado un nombre de método gestor de ese evento. Por ejemplo, el método **actionPerformed** es el encargado de gestionar eventos del tipo **ActionEvent**.

Cualquier clase que desee gestionar sus eventos debe implementar el interfaz (o interfaces) pensado para esa gestión y que determinará los procesos necesarios para gestionar el evento. En el caso de *ActionEvent* el interfaz es **ActionListener** que define como único método el ya comentado *actionPerformed*.

Es decir hay aquí tres actores fundamentales:

- El objeto de evento que se dispara cuando ocurre un suceso.
- El método de captura del evento

- El interfaz que tiene que estar implementado en la clase que desea capturar ese evento.

Sin duda, el más complejo es este último, pero hay que entender que un interfaz lo único que consigue es dar a una clase la facultad de escuchar (***Listen***) eventos.

Puede haber más de un método de captura de eventos, por ejemplo en el caso de eventos de foco (**FocusEvent**) hay dos métodos de manejo **focusGained** (cuando se obtiene el foco) y **focusLost** (cuando se pierde el foco) , ambos se define en la interfaz **FocusListener** que debe estar implementada en la clase que desea capturar los eventos.

fuentes de eventos

Aún falta definir al cuarto actor del manejo de eventos, y éste es la fuente que generará eventos. Esto es hacer que un objeto tenga capacidad de enviar eventos. Esto se realiza mediante un método que comienza por la palabra **add** seguida por el nombre del objeto que capturará los eventos.

Esto, dicho así, suena complicado. Pero es más fácil de lo que parece. Cualquier componente puede lanzar eventos, sólo hay que indicárselo, y eso es lo que hace el método add. En este método se pasa un único argumento que es el nombre de el objeto que manejará el evento. Ejemplo:

```
JButton boton1=new JButton("Prueba");

//Clase receptora de eventos
class Receptora implements ActionListener{
  ...
  void métodoLanzaEventos(){
        boton.addActionListener(this);//El botón lanza
              //eventos que esta misma clase maneja
  }
  ...
  public void actionPerformed(ActionEvent e){
        //Manejo del evento
  }
}
```

Hay un método **remove** que hace que un objeto deje de lanzar eventos. En el ejemplo:

```
boton.removeActionListener(this);
```

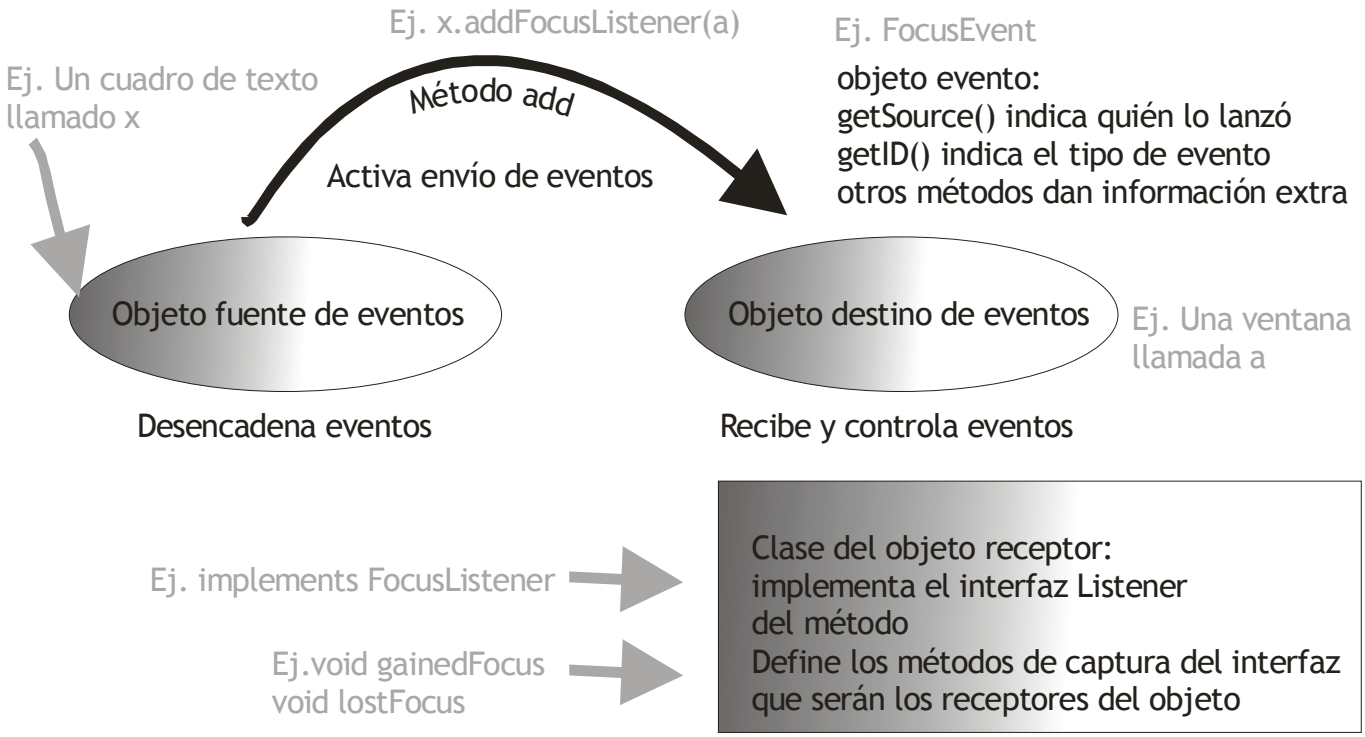


Ilustración 10, Proceso de gestión de eventos

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 **TooManyListenersException**

lista de eventos

Objeto de evento	**java.awt.event.ComponentEvent** Ocurre cuando un componente se mueva, cambia de tamaño o de visibilidad
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	**addComponentListener**
Interfaz de gestión	**ComponentListener**
Métodos de control	**componentResized** Cuando el componente cambió de tamaño **componentMoved** Cuando el componente se movió **componentShown** Cuando el componente se mostró **componentHidden** Cuando el componente cambió se puesooculto

Objeto de evento	**java.awt.event.FocusEvent** Ocurre cuando cambia el foco de un componente
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	**addFocusListener**
Interfaz de gestión	**FocusListener**
Métodos de control	**focusGained** Cuando se obtiene el foco **focusLost** Cuando se pierde el foco
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** (véase al final de la tabla)
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	**addKeyListener**
Interfaz de gestión	**KeyListener**
Métodos de control	**keyTyped** Cuando se pulsa una tecla que escribe (una tecla que escriba un símbolo Unicode, la tecla F1 por ejemplo no valdría) **keyPressed** Cuando se pulsa cualquier tecla **keyReleased** Cuando se levanta cualquier tecla
Métodos del objeto evento interesantes	**char getKeyChar()** Devuelve el código Unicode de la tecla pulsada **char 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 una tecla (y el componente posee el foco). Es un evento del tipo **InputEvent** (véase al final de la tabla)
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	**addMouseListener**
Interfaz de gestión	**MouseListener**
Métodos de control	**mouseClicked** Se hizo clic con un botón del ratón en el componente **mousePressed** Se pulsó un botón del ratón **mouseReleased** Se levantó un botón del ratón **mouseEntered** El ratón está sobre el componente. **mouseExited** El ratón salió fuera de el componente
Métodos del objeto evento 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
Objeto de evento	**java.awt.event.MouseMotionEvent** El mismo que el anterior. Pero en este caso se captura con el interfaz **MouseMotionListener** que está pensado para capturar eventos de movimiento de ratón.
Objetos que le pueden disparar	**Cualquier componente**
Método lanzador de eventos	**addMouseMotionListener**
Interfaz de gestión	**MouseMotionListener**
Métodos de control	**mouseDragged** Se arrastra con el ratón al componente **mouseMoved** Siempre que el ratón se mueva en el componente

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	**addContainerListener**
Interfaz de gestión	**ContainerListener**
Métodos de control	**componentAdded** Cuando se añade un componente **componentRemoved** 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	**addActionListener**
Interfaz de gestión	**ActionListener**
Métodos de control	**performedAction** Cuando se efectúa una acción
Métodos del objeto evento 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	**addAdjustmentListener**
Interfaz de gestión	**AdjustmentListener**
Métodos de control	**adjustmentValueChange** Cuando cambia el valor de una barra de desplazamiento
Métodos interesantes del objeto evento	**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

Objeto de evento	**javax.swing.event.HyperlinkEvent** Ocurre cuando le sucede algo a un texto de enlace (hipervínculo)
Objetos que le pueden disparar	**JEditorPane, JTextPane**
Método lanzador de eventos	**addHyperlinkListener**
Interfaz de gestión	**HyperlinkListener**
Métodos de control	**hyperlinkUpdate**
Métodos interesantes del objeto evento	***URL* getURL()** Devuelve la dirección URL del enlace

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	**addInternalFrameListener**
Interfaz de gestión	**InternalframeListener**
Métodos de control	**internalFrameActivated** Se activa la ventana **internalFrameClosed** Se cierra la ventana **internalFrameClosed** Se cierra la ventana **internalFrameDeactivated** Se desactiva la ventana **internalFrameDeiconified** Se restaura la ventana **internalFrameIconified** Se minimiza la ventana **internalFrameOpened** Se abre la ventana
Métodos interesantes del objeto evento	***InternalFrame* getInternalFrame()** Devuelve la ventana interna que originó el evento

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	**addItemListener**
Interfaz de gestión	**ItemListener**
Métodos de control	**itemStateChanged** Cuando se cambió un item
Métodos interesantes del objeto evento	***Object* getItem()** Obtiene el item 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	**addListSelectionListener**
Interfaz de gestión	**ListSelectionListener**
Métodos de control	**valueChanged** Cuando se cambió valor de la lista
Métodos interesantes del objeto evento	**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	**addMenuListener**
Interfaz de gestión	**MenuListener**
Métodos de control	**menuCanceled()** Si se canceló el menú **menuDeselected()** Si se deseleccionó el menú **menuSelected()** 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	**addPopupMenuListener**
Interfaz de gestión	**PopupMenuListener**
Métodos de control	**popupMenuCanceled()** Si se canceló el menú **popupMenuWillBecomeInvisible()** Si va a desaparecer el menú **popupMenuWillBecomeVisible()** Si va a aparecer el menú

Objeto de evento	**javax.swing.event.MenuKeyEvent** Ocurre cuando se arrastró pulsaron teclas hacia el menú
Objetos que le pueden disparar	**JMenuItem**
Método lanzador	**addMenuKeyListener**
Interfaz de gestión	**MenuKeyListener**
Métodos de control	**popupMenuKeyPressed()** **popupMenuKeyReleased()** **popupMenuKeyTyped()**

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	**addMenuDragMouseListener**
Interfaz de gestión	**MenuDragMouseListener**
Métodos de control	**popupMenuDragMouseDragged()** El ratón está arrastrando sobre el menú **popupMenuDragMouseEntered()** El ratón está arrastrando dentro del menú **popupMenuDragMouseExited()** El ratón está arrastrando hacia fuera del menú **popupMenuDragMouseReleased()** Se soltó el ratón que se arrastraba dentro del menú

Objeto de evento	**java.awt.event.WindowEvent** Captura eventos de ventana
Objetos que le pueden disparar	**JDialog, JWindow, JFrame**
Método lanzador	**addWindowListener**
Interfaz de gestión	**WindowListener**
Métodos de control	**windowOpened()** La ventana se está abriendo **windowClosing()** La ventana se está cerrando **windowClosed()** La ventana se cerró **windowIconified()** Se minimizó la ventana **windowDeiconified()** Se restauró la ventana **windowActivated()** Se activó la ventana

eventos InputEvent

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) no 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.

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**

apariencia

root panes

Los paneles son elementos interiores a los contenedores que gestionan su contenido. 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 tres objetos: son el *glass pane*, el *content pane* y el *layered pane*.

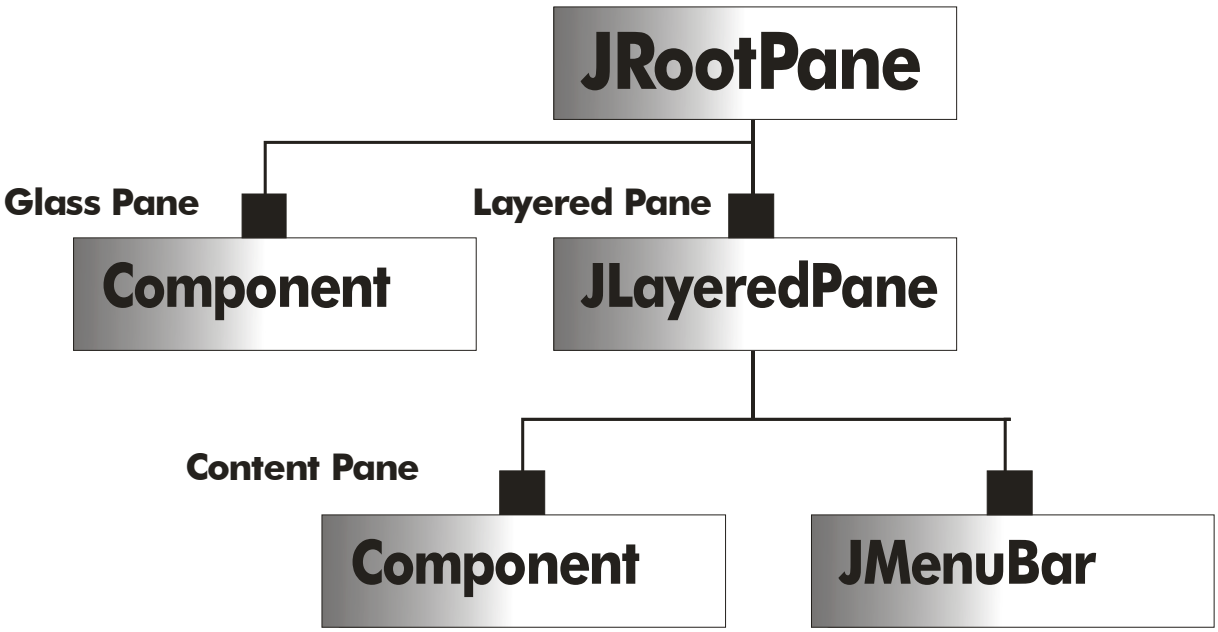


Ilustración 11, Estructura de los paneles de las ventanas

layered pane

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):

- DEFAULT_LAYER. Capa en la que se colocan los componentes. Es la capa estándar y la que está más abajo.
- PALETTE_LAYER. Capa de paletas. Pensada para barras de herramientas y paletas flotantes.
- MODAL_LAYER. Cuadros de diálogo modales.
- POPUP_LAYER. Capa para el menú emergente.
- DRAG_LAYER. Para realizar arrastre con el ratón.