introducción

historia de Java

los antecedentes de Java

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

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

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

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

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

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

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

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

Otras adaptaciones famosas fueron:

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

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

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

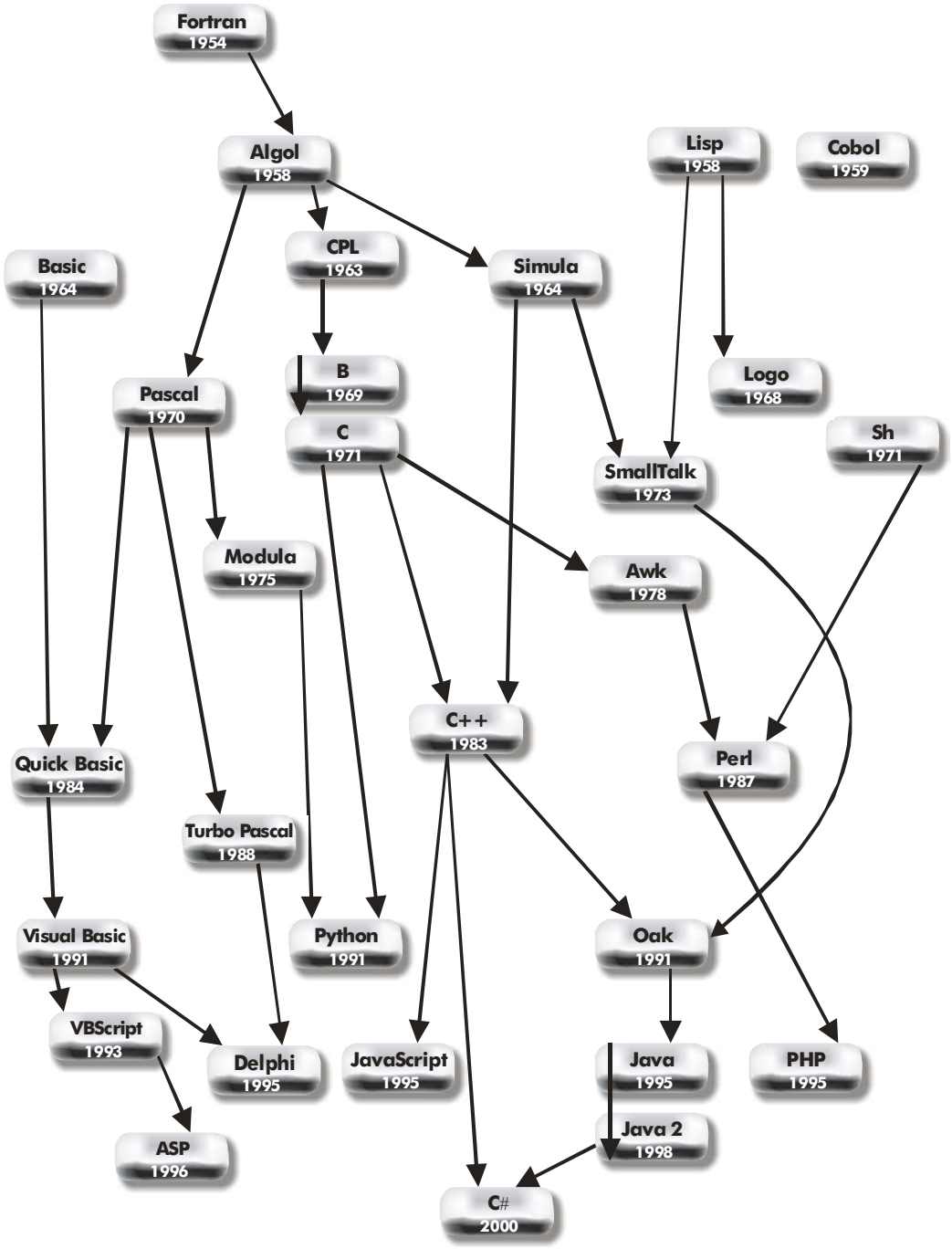


Ilustración 1, ***Evolución de algunos lenguajes de programación***

la llegada de Java

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

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

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

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

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

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

Java y JavaScript

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

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

características de Java

bytecodes

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

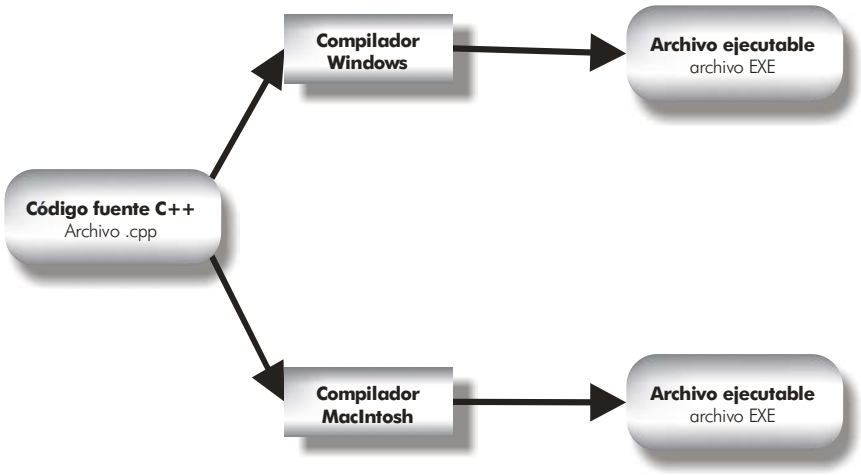


Ilustración 2, ***Proceso de compilación de un programa C++***

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

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

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

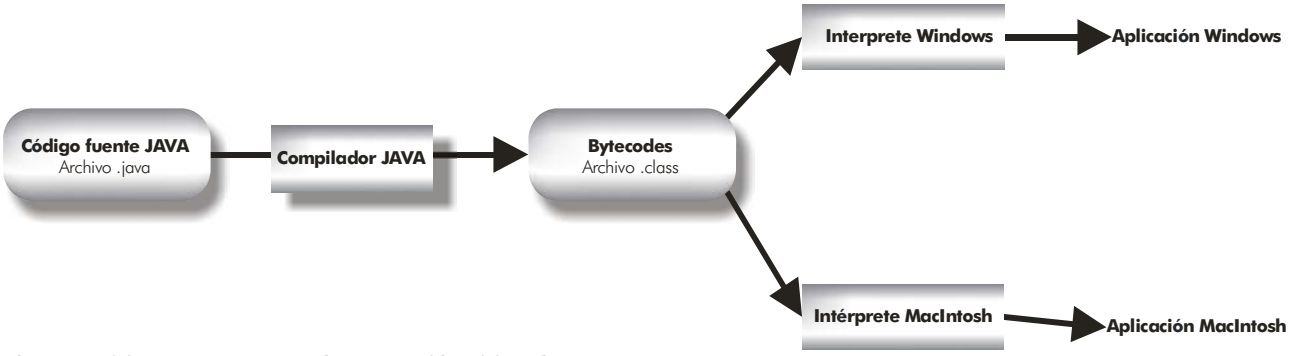


Ilustración 3, ***Proceso de compilación de un programa Java***

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

seguridad

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

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

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

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

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

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

tipos de aplicaciones Java

applet

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

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

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

aplicaciones de consola

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

aplicaciones gráficas

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

servlets

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

empezar a trabajar con Java

el kit de desarrollo Java (JDK)

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

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

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

versiones de Java

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

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

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

Java 1.0

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

Java 1.1

Mejoró la versión anterior incorporando las siguientes mejoras:

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

Java 2

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

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

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

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

Java 1.3 y 1.4

Son las últimas versiones de Java2 que incorporan algunas mejoras,

plataformas

Actualmente hay tres ediciones de la plataforma Java 2

J2SE

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

J2EE

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

J2ME

Pensada para la creación de aplicaciones Java para dispositivos móviles.

entornos de trabajo

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

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

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

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

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

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

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

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

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

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

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

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

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

escritura de programas Java

codificación del texto

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

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

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

notas previas

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

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

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

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

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

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

el primer programa en Java

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

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

proceso de compilación

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

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

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

```
javadoc archivo.java
```

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

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

```
java archivoClass
```

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

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

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

2> Ir al apartado **Opciones avanzadas**

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

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

Ejemplo de contenido de la variable path:

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

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

javadoc

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

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

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

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

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

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

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

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

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

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

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

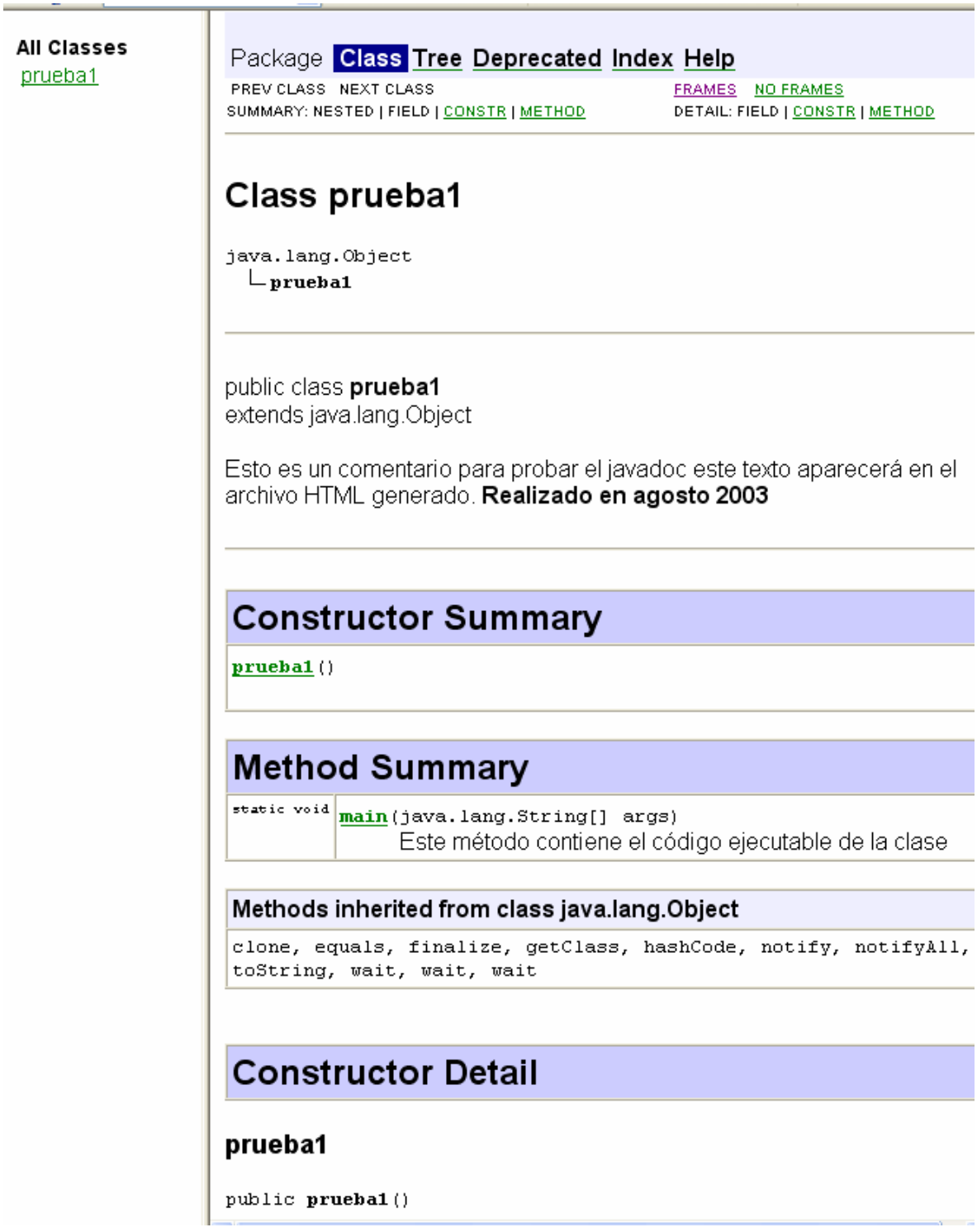
Ejemplo:

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

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

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

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



All Classes
prueba1

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class prueba1

java.lang.Object
└ prueba1

public class **prueba1**
extends java.lang.Object

Esto es un comentario para probar el javadoc este texto aparecerá en el archivo HTML generado. **Realizado en agosto 2003**

Constructor Summary

prueba1()

Method Summary

static void	**main**(java.lang.String[] args) Este método contiene el código ejecutable de la clase

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

prueba1

public **prueba1**()

Ilustración 4, Página de documentación de un pro grama Java

instrucción import

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

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

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

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

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

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

También se puede utilizar el asterisco en esta forma:

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

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

variables

introducción

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

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

declaración de variables

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

```
tipo nombrevariable;
```

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

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

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

```
int dias=365;
```

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

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

Al declarar una variable se puede incluso utilizar una expresión:

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

alcance o ámbito

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

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

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

tipos de datos primitivos

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

enteros

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

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

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

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

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

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

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

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

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

números en coma flotante

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

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

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

booleanos

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

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

caracteres

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

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

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

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

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

conversión entre tipos *(casting)*

Hay veces en las que se deseará realizar algo como:

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

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

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

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

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

En el siguiente ejemplo:

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

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

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

operadores

introducción

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

operadores aritméticos

Son:

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

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

Por ejemplo:

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

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

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

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

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

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

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

operadores condicionales

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

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

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

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

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

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

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

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

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

operadores de BIT

Manipulan los bits de los números. Son:

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

operadores de asignación

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

```
x += 3;
```

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

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

También se pueden concatenar asignaciones:

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

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

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

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

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

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

operador ?

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

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

Ejemplo:

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

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

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

precedencia

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

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

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

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

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

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

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

Esto último provoca conflictos, por ejemplo en:

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

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

Una vez más los paréntesis podrían evitar estos conflictos.

la clase Math

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

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

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

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

Math posee dos constantes, que son:

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

Por otro lado posee numerosos métodos que son:

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

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

estructuras de control del flujo

if

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

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

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

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

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

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

switch

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

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

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

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

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

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

Ejemplo 1:

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

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

Ejemplo 2:

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

while

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

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

Sintaxis:

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

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

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

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

do while

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

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

for

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

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

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

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

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

Ejemplo (factorial):

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

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

sentencias de salida de un bucle

break

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

continue

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

arrays y cadenas

arrays

unidimensionales

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

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

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

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

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

```
tipo nombre[];
```

Ejemplo:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

arrays multidimensionales

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

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

notas es un array que contiene arrays de enteros

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

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

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

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

Se pueden utilizar más de dos dimensiones si es necesario.

longitud de un array

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

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

la clase Arrays

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

Su uso es

```
Arrays.método(argumentos);
```

fill

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

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

También permite decidir desde que índice hasta qué índice rellenamos:

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

equals

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

sort

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

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

binarySearch

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

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

el método System.arraysCopy

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

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

clase String

introducción

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

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

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

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

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

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

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

comparación entre objetos String

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

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

String.valueOf

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

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

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

métodos de las variables de las cadenas

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

length

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

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

concatenar cadenas

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

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

charAt

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

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

substring

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

```
String s1="Buenos días";
```

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

indexOf

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

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

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

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

lastIndexOf

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

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

También permite comenzar a buscar desde una determinada posición.

endsWith

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

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

startsWith

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

replace

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

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

replaceAll

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

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

toUpperCase

Devuelve la versión en mayúsculas de la cadena.

toLowerCase

Devuelve la versión en minúsculas de la cadena.

toCharArray

Obtiene un array de caracteres a partir de una cadena.

lista completa de métodos

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

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

objetos y clases

programación orientada a objetos

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

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

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

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

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

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

propiedades de la POO

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

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

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

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

introducción al concepto de objeto

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

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

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

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

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

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

clases

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

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

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

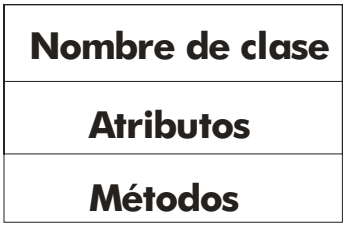


Ilustración 5, Clase en notación UML

El formato general para crear una clase en Java es:

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

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

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

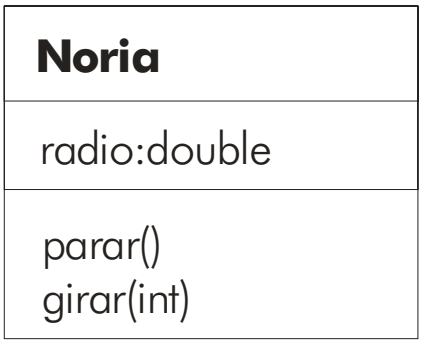


Ilustración 6, Clase Noria bajo notación UML

objetos

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

datos miembro (propiedades o atributos)

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

```
objeto.atributo
```

Por ejemplo:

```
Noria.radio;
```

métodos

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

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

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

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

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

herencia

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

creación de objetos de la clase

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

```
Noria noriaDePalencia;
```

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

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

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

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

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

especificadores de acceso

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

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

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

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

creación de clases

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

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

Ejemplo:

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

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

Las variables locales de una clase pueden ser inicializadas.

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

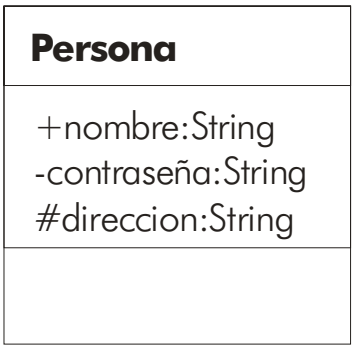


Ilustración 8, La clase persona en UML. El signo + significa *public*, el signo # *protected* y el signo - *private*

definir métodos de clase (operaciones o funciones de clase)

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

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

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

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

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

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

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

Ejemplo:

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

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

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

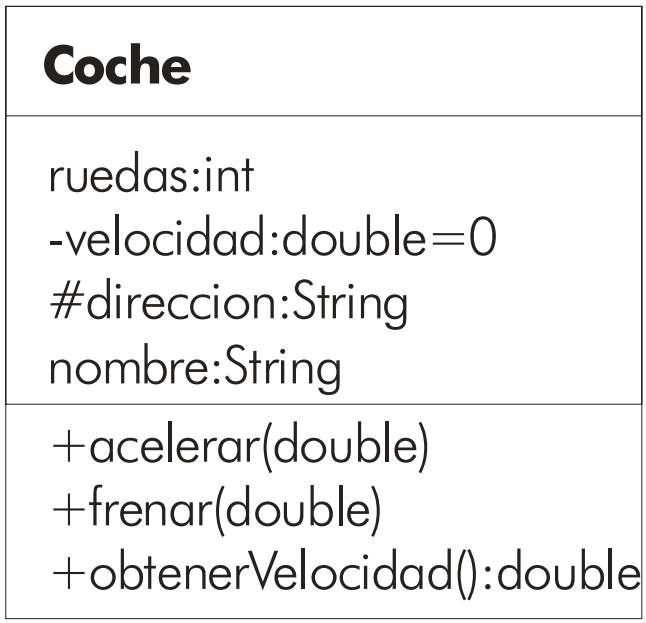


Ilustración 9, Versión UML de la clase Coche

argumentos por valor y por referencia

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

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

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

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

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

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

Sin embargo en este otro caso:

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

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

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

devolución de valores

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

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

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

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

sobrecarga de métodos

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

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

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

la referencia *this*

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

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

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

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

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

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

Los posibles usos de **this** son:

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

creación de constructores

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

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

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

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

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

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

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

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

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

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

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

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

métodos y propiedades genéricos (*static*)

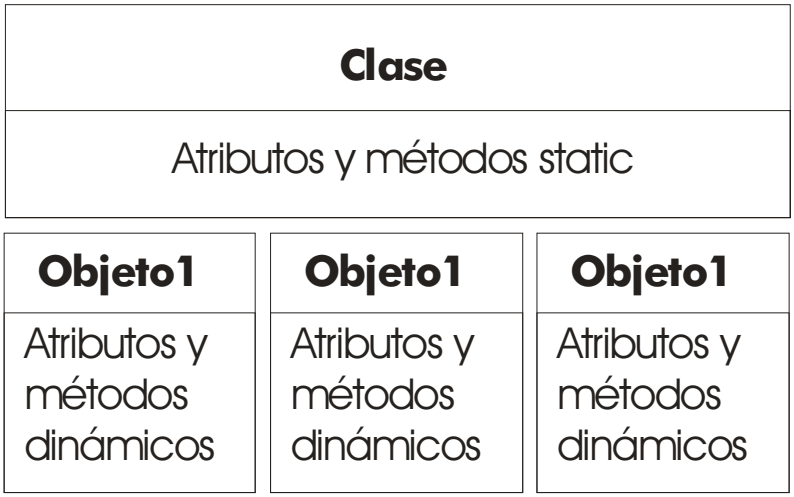


Ilustración 10, *Diagrama de funcionamiento de los métodos y atributos static*

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

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

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

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

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

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

el método main

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

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

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

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

```
java claseConMain uno dos
```

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

destrucción de objetos

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

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

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

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

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

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

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

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

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

public class app {
```

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

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

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

el método *finalize*

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

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

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

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

reutilización de clases

herencia

introducción

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

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

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

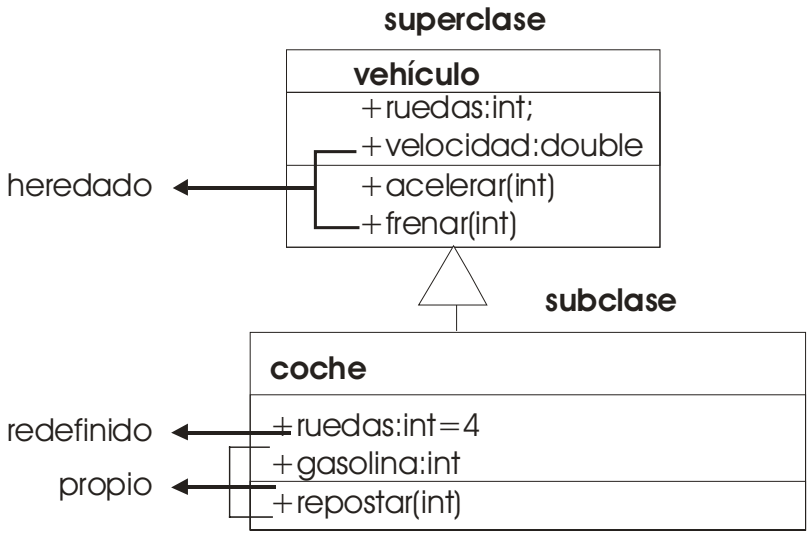


Ilustración 11, herencia

métodos y propiedades no heredados

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

Ejemplo:

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

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

anulación de métodos

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

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

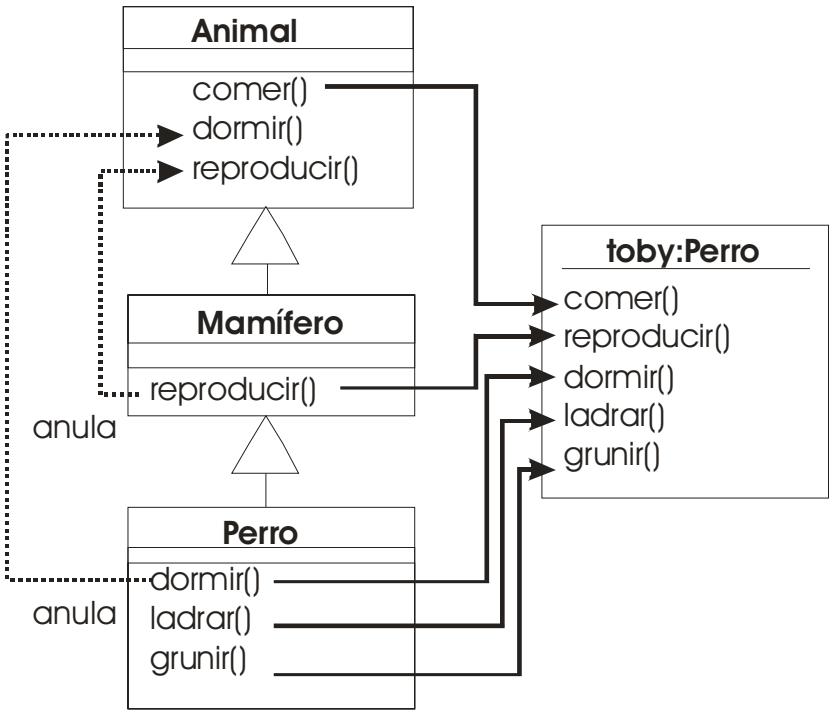


Ilustración 12, anulación de métodos

super

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

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

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

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

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

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

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

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

Por defecto Java realiza estas acciones:

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

casting de clases

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

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

Ejemplo:

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

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

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

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

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

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

instanceof

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

```
objeto instanceof clase
```

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

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

En el ejemplo anterior aparecerá en pantalla:

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

clases abstractas

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

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

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

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

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

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

final

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

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

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

clases internas

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

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

Por ejemplo:

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

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

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

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

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

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

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

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

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

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

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

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

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

interfaces

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

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

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

utilizar interfaces

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

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

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

creación de interfaces

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

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

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

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

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

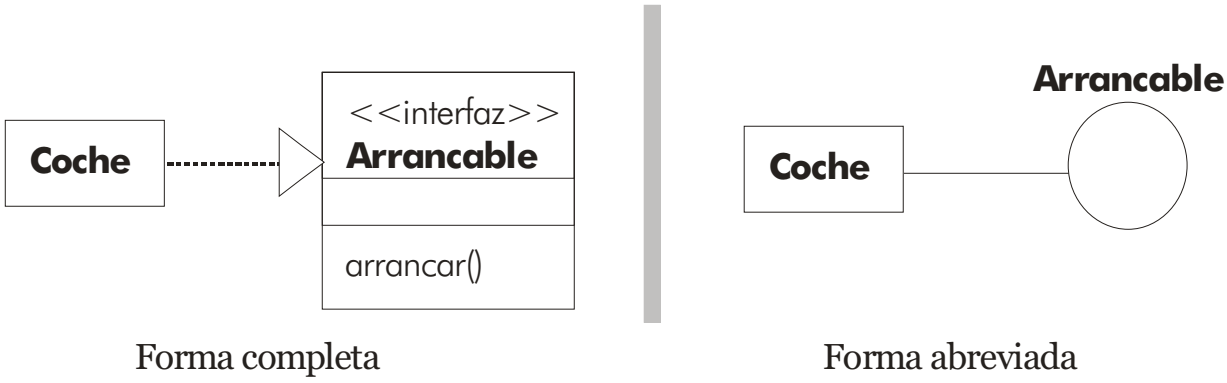


Ilustración 13, Diagramas de clases UML sobre la interfaz *Arrancable*

subinterfaces

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

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

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

variables de interfaz

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

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

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

interfaces como funciones de retroinvocación

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

Ejemplo:

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

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

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

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

creación de paquetes

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

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

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

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

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

Ejemplo:

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

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

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

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

Es decir:

```
paquete.clase
```

O incluso:

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

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

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

organización de los paquetes

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

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

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

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

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

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

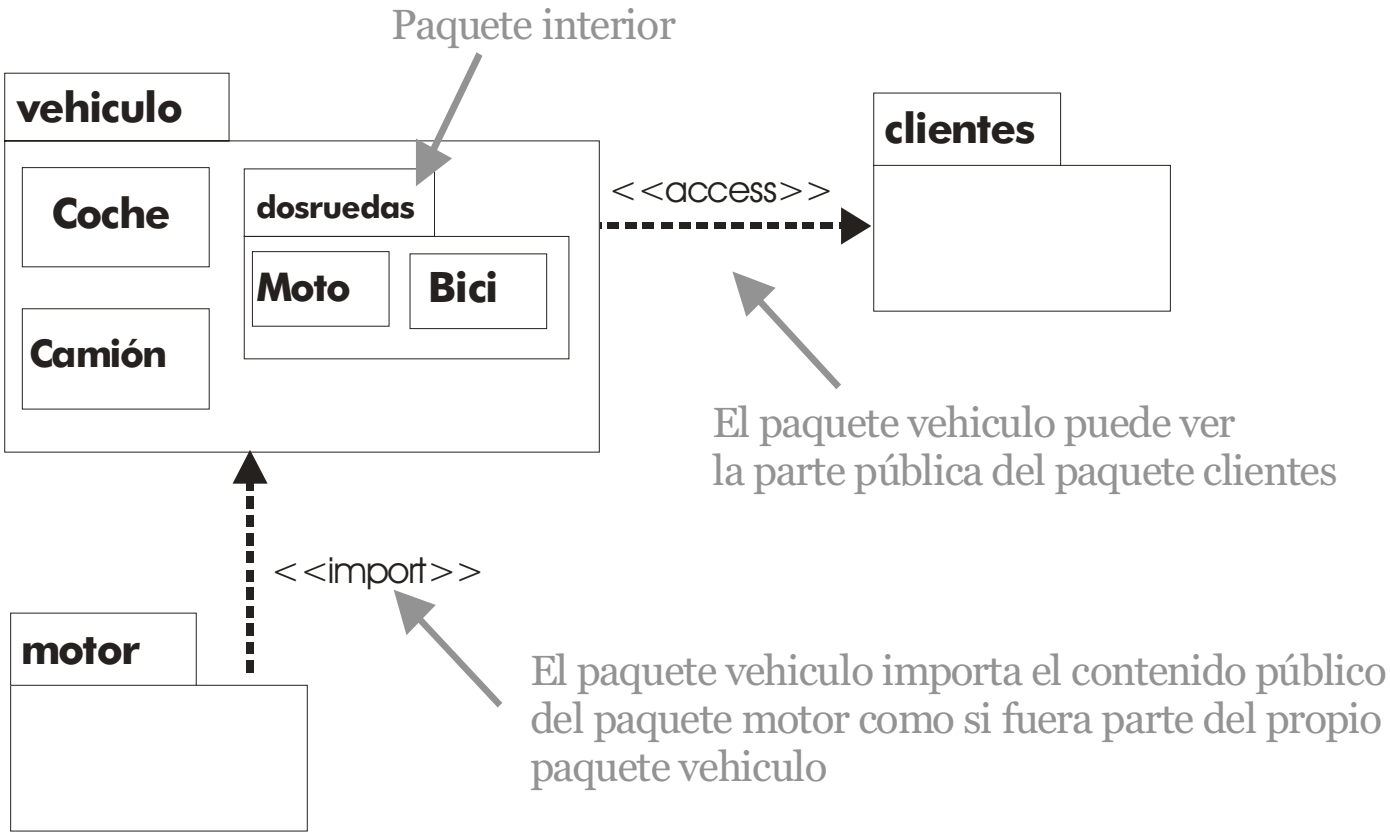


Ilustración 14, diagrama de paquetes UML

excepciones

introducción a las excepciones

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

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

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

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

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

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

try y catch

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

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

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

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

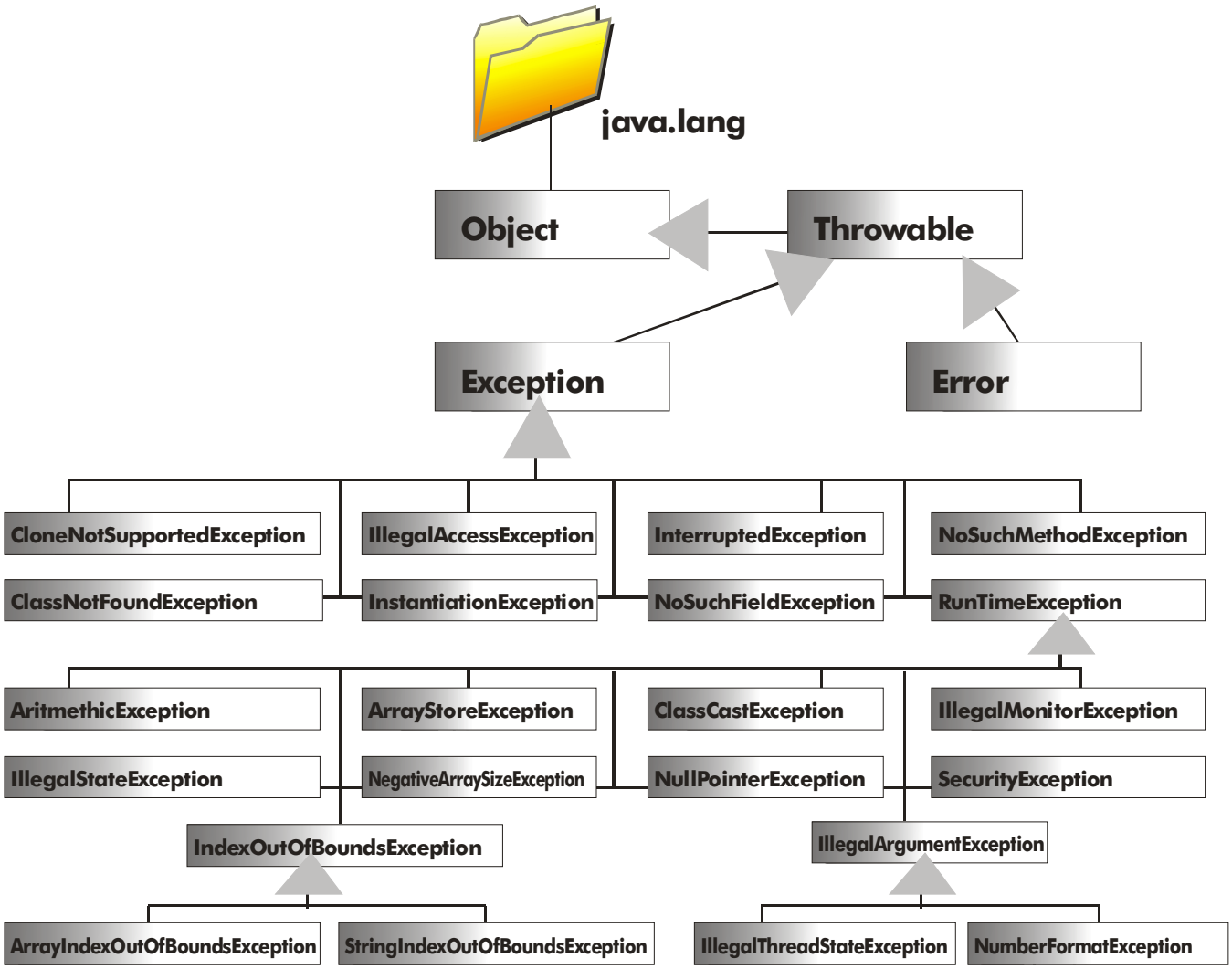


Ilustración 15, Jerarquía de las clases de manejo de excepciones

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

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

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

manejo de excepciones

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

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

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

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

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

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

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

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

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

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

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

throws

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

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

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

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

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

Ejemplo:

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

throw

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

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

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

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

Eso construye una excepción con el mensaje indicado.

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

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

El segundo catch capturará también las excepciones del primer tipo

finally

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

```
try{
   ...
```

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

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

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

clases fundamentales (I)

la clase *Object*

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

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

comparar objetos

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

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

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

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

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

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

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

código hash

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

clonar objetos

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

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

Ejemplo:

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

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

método toString

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

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

Si redefinimos este método en la clase Coche:

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

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

lista completa de métodos de la clase Object

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

clase *Class*

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

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

Ejemplo:

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

lista de métodos de *Class*

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

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

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

reflexión

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

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

clase Field

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

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

clase Method

Representa métodos de una clase. Sus propios métodos son:

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

clase Constructor

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

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

clases para tipos básicos

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

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

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

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

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

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

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

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

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

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

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

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

clase StringBuffer

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

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

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

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

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

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

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

métodos de StringBuffer

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

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

números aleatorios

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

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

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

métodos de Random

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

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

fechas

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

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

clase Calendar

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

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

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

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

clase GregorianCalendar

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

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

método get

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

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

método set

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

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

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

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

método getTime

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

método setTime

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

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

método getTimeInMillis

Devuelve el número de milisegundos que representa esa fecha.

clase Date

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

construcción

Hay varias formas de crear objetos Date:

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

método after

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

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

método before

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

métodos equals y compareTo

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

clase DateFormat

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

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

creación básica

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

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

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

el método format

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

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

creaciones de formato sofisticadas

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

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

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

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

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

Ejemplo:

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

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

método parse

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

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

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

Es un método muy poco usado.

cadenas delimitadas. *StringTokenizer*

introducción

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

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

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

construcción

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

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

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

USO

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

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

Ejemplo:

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

entrada y salida en Java

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

clases para la entrada y la salida

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

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

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

InputStream/ OutputStream

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

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

Reader/Writer

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

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

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

InputStreamReader/ OutputStreamWriter

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

DataInputStream/DataOutputStream

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

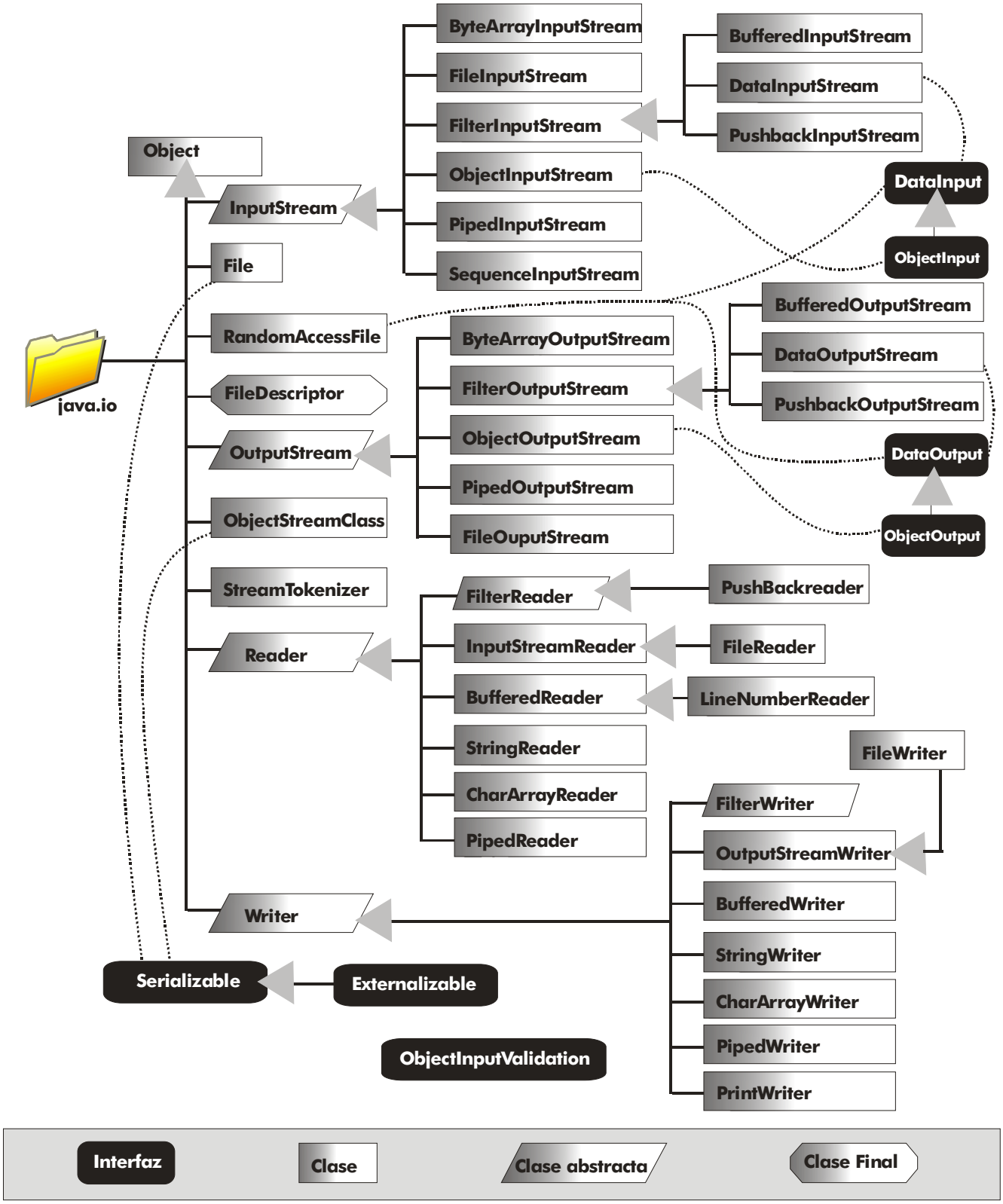


Ilustración 16, Clases e interfaces del paquete java.io

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

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

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

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

Por su parte, los métodos de DataOutputStream son:

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

ObjectInputStream/ObjectOutputStream

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

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

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

BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter

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

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

PrintWriter

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

FileInputStream/FileOutputStream/FileReader/FileWriter

Leen y escriben en archivos (File=Archivo).

PipedInputStream/PipedOutputStream

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

entrada y salida estándar

las clases in y out

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

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

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

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

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

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

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

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

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

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

Conversión a forma de Reader

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

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

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

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

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

Lectura con readLine

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

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

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

Ficheros

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

clase File

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

construcción de objetos de archivo

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

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

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

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

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

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

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

el problema de las rutas

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

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

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

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

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

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

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

métodos generales

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

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

métodos de carpetas

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

métodos de archivos

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

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

secuencias de archivo

lectura y escritura byte a byte

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

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

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

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

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

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

escritura

El proceso sería:

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

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

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

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

Ejemplo:

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

lectura

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

Ejemplo, leer los números del ejemplo anterior :

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

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

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

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

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

lectura y escritura mediante caracteres

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

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

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

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

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

Ejemplo:

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

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

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

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

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

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

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

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

RandomAccessFile

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

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

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

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

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

Los métodos fundamentales son:

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

el administrador de seguridad

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

serialización

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

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

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

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

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

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

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

clases fundamentales (II)
colecciones

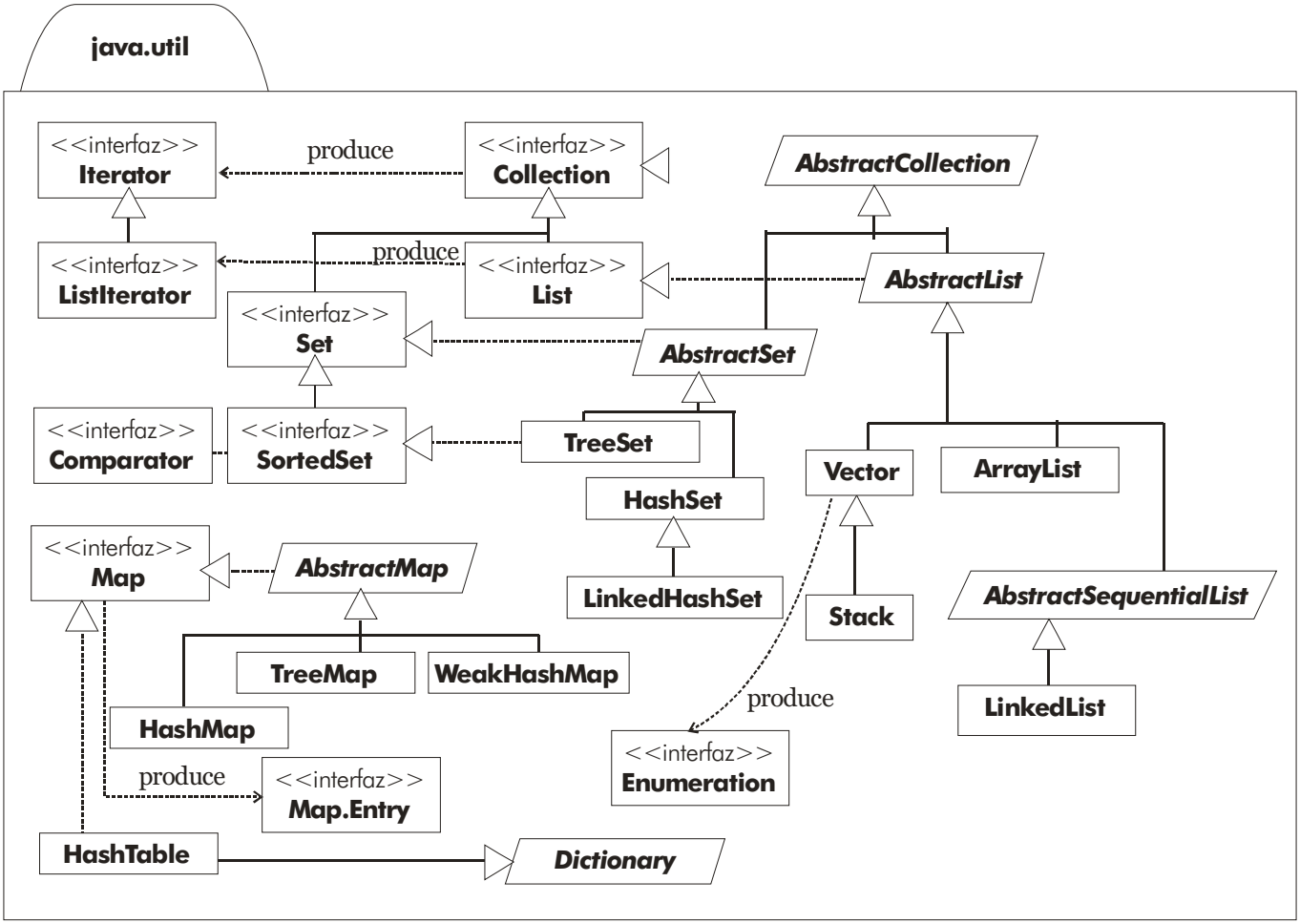


Ilustración 17, Diagrama de clases UML de las clases de colecciones

estructuras estáticas de datos y estructuras dinámicas

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

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

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

interfaz Collection

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

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

iteradores

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

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

Ejemplo (recorrido por una colección):

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

Listas enlazadas

interfaz *List*

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

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

ListIterator

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

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

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

clase *ArrayList*

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

Posee tres constructores:

- **ArrayList().** Constructor por defecto. Simplemente crea un ArrayList vacío
- **ArrayList(int** capacidadInicial**).** Crea una lista con una capacidad inicial indicada.
- **ArrayList(Collection** c**).** Crea una lista a partir de los elementos de la colección indicada.

clase LinkedList

Crea listas de adición doble (desde el principio y el final). Implementa la interfaz *List*. Desde este clase es sencillo implantar estructuras en forma de pila o de cola. Añade los métodos:

método	uso
Object getFirst()	Obtiene el primer elemento de la lista
Object getLast()	Obtiene el último elemento de la lista
void addFirst(Object o**)**	Añade el objeto al principio de la lista
void addLast(Object o**)**	Añade el objeto al final de la lista
void removeFirst()	Borra el primer elemento
void removeLast()	Borra el último elemento

colecciones sin duplicados

interfaz Set

Define métodos para crear listas dinámicas de elementos sin duplicados. Deriva de *Collection*. Es el método **equals** el que se encarga de determinar si dos objetos son duplicados en la lista (habrá que redefinir este método para que funcione adecuadamente).

Posee los mismos métodos que la interfaz **Collection**. La diferencia está en el uso de duplicados.

clase *HashSet*

Implementa la interfaz anterior.

árboles. *SortedSet*

Un árbol es una colección ordenada de elementos. Al recorrer esta estructura, los datos aparecen automáticamente en el orden correcto. La adición de elementos es más lenta, pero su recorrido ordenado es mucho más eficiente.

La interfaz **SortedSet** es la encargada de definir esta estructura. Esta interfaz deriva de **Collection** y añade estos métodos:

método	uso
Object **first()**	Obtiene el primer elemento del árbol (el más pequeño)
Object **last()**	Obtiene el último elemento del árbol (el más grande)
SortedSet **headSet(**Object o**)**	Obtiene un SortedSet que contendrá todos los elementos menores que el objeto *o*.
SortedSet **tailSet(**Object o**)**	Obtiene un SortedSet que contendrá todos los elementos mayores que el objeto *o*.
SortedSet **subSet(**Object menor, Object mayor**)**	Obtiene un SortedSet que contendrá todos los elementos del árbol cuyos valores ordenados estén entre el menor y mayor objeto indicado
Comparator **comparator()**	Obtiene el objeto comparador de la lista

El resto de métodos son los de la interfaz Collection (sobre todo add y remove). La clase **TreeSet** implementa esta interfaz.

comparaciones

El problema es que los objetos tienen que poder ser comparados para determinar su orden en el árbol. Esto implica implementar la interfaz **Comparable** de Java (está en **java.lang**). Esta interfaz define el método **compareTo** que utiliza como argumento un objeto a comparar y que devuelve 0 si los objetos son iguales, 1 si el primero es mayor que el segundo y -1 en caso contrario.

Con lo cual los objetos a incluir en un TreeSet deben implementar Comparator y esto les obliga a redefinir el método compareTo (recordando que su argumento es de tipo ***Object*).**

Otra posibilidad es utilizar un objeto **Comparator**. Esta es otra interfaz que define el método **compare** al que se le pasan dos objetos. Su resultado es como el de *compareTo* (0 si son iguales, 1 si el primero es mayor y -1 si el segundo es mayor). Para que un árbol utilice este tipo de objetos se les pasa como argumentos en su creación.

mapas

Permiten definir colecciones de elementos que poseen pares de datos clave-valor. Esto se utiliza para localizar valores en función de la clave que poseen. Son muy interesantes y rápidos. Es la nueva implementación de tablas hash (ver tablas hash, más adelante). Métodos:

método	uso
Object get(Object clave**)**	Devuelve el objeto que posee la clave indicada
Object put(Object clave, **Object** valor**)**	Coloca el par clave-valor en el mapa. Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null**
Object remove(Object clave**)**	Elimina de la lista el valor asociado a esa clave.
boolean containsKey(Object clave**)**	Indica si el mapa posee la clave señalada
boolean containsValue(Object valor**)**	Indica si el mapa posee el valor señalado
void putAll(Map mapa**)**	Añade todo el mapa al mapa actual
Set keySet()	Obtiene un objeto *Set* creado a partir de las claves del mapa
Collection values()	Obtiene la colección de valores del mapa
Set entrySet()	Devuelve una lista formada por objetos **Map.Entry**

El objeto **Map.Entry** es interno a los objetos **Map** y representa un objeto de par clave/valor. Tiene estos métodos:

método	uso
Object getKey()	Obtiene la clave del elemento actual **Map.Entry**
Object getValue()	Obtiene el valor
Object setValue(Object valor**)**	Cambia el valor y devuelve el valor anterior del objeto actual

Esta interfaz está implementada en la clase **HashMap**. Además existe la interfaz **SortedMap** implementada en **TreeMap.** La diferencia es que *TreeMap* crea un árbol ordenado con las claves (el manejo es el mismo).

colecciones de la versión 1.0 y 1.1

En las primeras versiones de Java, el uso de las colecciones estaba restringido a las clases **Vector** y **Stack**. Desde Java 2 se anima a no utilizarlas, aunque se las mantiene por compatibilidad.

clase *Vector*

La clase **Vector** implementa la interfaz *List*. Es una clase veterana casi calcada a la clase *ArrayList*. En las primeras versiones de Java era la única posibilidad de implementar arrays dinámicos. Actualmente sólo se recomienda su uso si se utiliza una estructura dinámica para usar con varios *threads*.

Esto se debe a que esta clase implementa todos los métodos con la opción **synchronized**. Como esta opción hace que un método se ejecute más lentamente, se recomienda suplantar su uso por la clase *ArrayList* en los casos en los que la estructura dinámica no requiera ser sincronizada.

Otra diferencia es que permite utilizar la interfaz **Enumeration** para recorrer la lista de vectores. Las variables **Enumeration** tienen dos métodos **hasMoreElements** que indica si el vector posee más elementos y el método **nextElement** que devuelve el siguiente el elemento del vector (si no existiera da lugar a la excepción **NoSuchElementException**). La variable Enumeration de un vector se obtiene con el método **Elements** que devuelve una variable **Enumeration.**

clase Stack

Es una clase derivada de la anterior usada para crear estructuras de pilas. Las pilas son estructuras dinámicas en las que los elementos se añaden por arriba y se obtienen primero los últimos elementos añadidos. Sus métodos son:

método	uso
Object **push(**Object elemento**)**	Coloca un nuevo elemento en la pila
Object **pop()**	Retira el último elemento añadido a la pila. Si la pila está vacía, causa una excepción de tipo **EmptyStackException** (derivada de **RuntimeException**).
Object **peek()**	Obtiene el último elemento de la pila, pero sin retirarlo.

clase abstracta Dictionary

La desventaja de las estructuras anteriores reside en que su manipulación es larga debido a que el orden de la lista permanece en todo momento, lo que obliga a recorrer la lista elemento a elemento.

La clase abstracta Dictionary proporciona la interfaz para trabajar con mapas de valores clave. Actualmente está absolutamente reemplazado con la interfaz Map. La idea es proporcionar estructuras con un par de valores, código - contenido. En estas estructuras se busca el código para obtener el contenido.

tablas Hash

La clase **HashTable** implementa la clase anterior y proporciona métodos para manejar la estructura de los datos. Actualmente está en desuso ya que hay clases más poderosas.

En estas tablas cada objeto posee un código hash que es procesado rápidamente. Los elementos que tengan el mismo código hash, forman una lista enlazada. Con lo cual una tabla hash es una lista de listas enlazadas, asociadas con el mismo código.

Cada elemento a añadir en una tabla hash posee una clave y un valor. Ambos elementos son de tipo **Object**. La clave está pensada para ser buscada de forma rápida. La idea es que a partir de la clave obtenemos el objeto.

En las tablas hash es fundamental la obtención del código hash de cada objeto. Esto lo realiza el método **hashCode** que permite conocer el código hash de un objeto. Este método está implementado en la clase *Object*, pero a veces hay que redefinirlo en las clases de usuario para que funcione de manera conveniente. En cualquier caso, con los mismos datos, el algoritmo hashCode, obtiene el mismo código.

No obstante el método **hasCode** se puede redefinir para calcular el código de la forma que se estime conveniente.

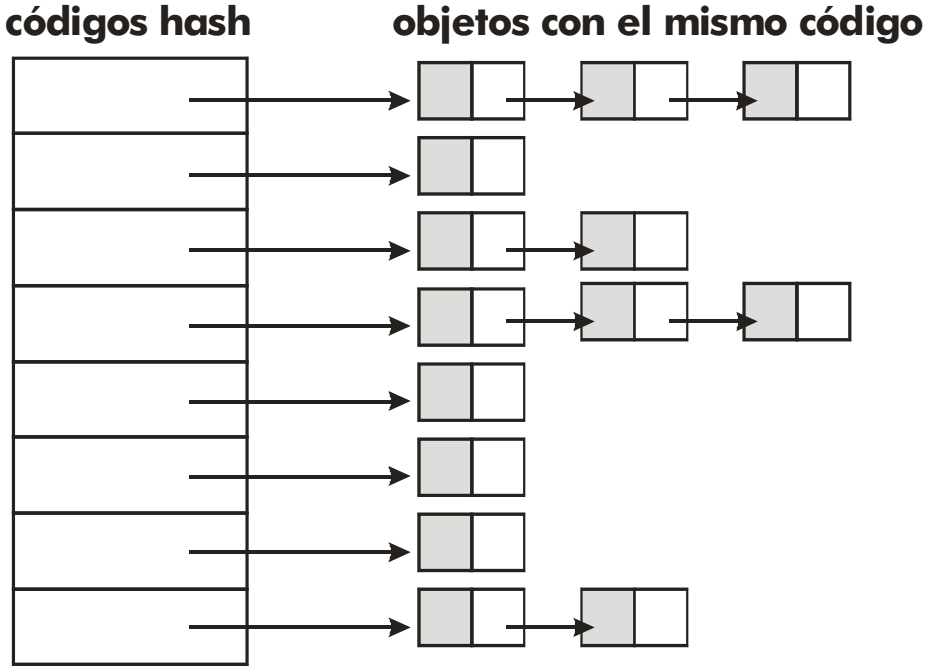


Ilustración 18, Estructura de las tablas hash

Métodos de **Hashtable**:

método	uso
Object **put(**Object clave, Object elemento**)**	Asocia la clave indicada al elemento. Devuelve excepción **NullPointerException** si la clave es nula. Devuelve el valor previo para esa misma clave
Object **get(**Object key)	Obtiene el valor asociado a esa clave.
Object **remove(**Object key)	Elimina el valor asociado a la clave en la tabla hash. Además devuelve ese valor
int **size()**	Obtiene el número de claves de la tabla hash
Enumeration **keys()**	Obtiene un objeto de enumeración para recorrer las claves de la tabla

método	uso
Enumeration **element()**	Obtiene un objeto de enumeración para recorrer los valores de la tabla
Set **keySet()**	(versión 1.2) Obtiene un objeto **Set** con los valores de las claves
Set **entrySet()**	(versión 1.2) Obtiene un objeto **Set** con los valores de la tabla
boolean **containsKey(**Object key)	**true** si la clave está en la tabla
boolean **containsValue(**Object valor)	**true** si el valor está en la tabla

la clase *Collections*

Hay una clase llamada **Collections** (no confundir con la interfaz **Collection**) que permite obtener fácilmente colecciones especiales, esto es lo que se conoce como envoltorio. Sus métodos son:

método	uso
static Collection **synchronizedCollection(**Collection c**)**	Obtiene una colección con métodos sincronizados a partir de la colección c
static List **synchronizedList(**List c**)**	Obtiene una lista con métodos sincronizados a partir de la lista c
static Set **synchronizedSet(**Set c**)**	Obtiene una tabla hash sincronizada a partir de la tabla hash c
static Set **synchronizedSortedSet(** SortedSet c**)**	Obtiene un árbol sincronizado a partir del árbol c
static Map **synchronizedMap(**Map c**)**	Obtiene un mapa sincronizado a partir del mapa c
static SortedMap **synchronizedSortedMap(**SortedMap c**)**	Obtiene un mapa ordenado sincronizado a partir de c
static Collection **unmodifiableCollection(**Collection c**)**	Obtiene una colección de sólo lectura a partir de la colección c
static List **unmodifiableList(**List c**)**	Obtiene una lista de sólo lectura a partir de la lista c
static Set **unmodifiableSet(**Set c**)**	Obtiene una tabla hash de sólo lectura a partir de la tabla hash c
static Set **unmodifiableSortedSet(** SortedSet c**)**	Obtiene un árbol de sólo lectura a partir de el árbol c
static Map **unmodifiableMap(**Map c**)**	Obtiene un mapa de sólo lectura a partir del mapa c
static SortedMap **unmodifiableSortedMap(**SortedMap c**)**	Obtiene un mapa ordenado de sólo lectura a partir de c
static void **sort(**List l**)**	Ordena la lista
static void **sort(**List l, Comparator c**)**	Ordena la lista basándose en el comparador indicado

método	uso
static int binarySearch(List l, **Object** o**)**	Busca de forma binaria el objeto en la lista (la lista tiene que estar ordenada en ascendente)
static int binarySearch(List l, **Object** o, **Comparator** *c***)**	Busca de forma binaria el objeto en la lista. La lista tiene que estar ordenada en ascendente usando el objeto comparador *c*

clases fundamentales (y III)

números grandes

Cuando se manipulan números sobre los que se requiere gran precisión, los tipos estándar de Java (**int, long, double,** etc.) se quedan cortos. Por ello en el paquete **java.math** disponemos de dos clases dedicadas a la precisión de números.

clase BigInteger

Se utiliza para cuando se desean almacenar números que sobrepasan los 64 bits del tipo **long**.

creación

constructor	uso
BigInteger(String texto**)** **throws** NumberFormatException	Crea un objeto para enteros grandes usando el número representado por el texto. En el caso de que el número no sea válido se lanza una excepción **NumberFormatException**
BigInteger(String texto, int base**)** **throws** NumberFormatException	Constructor idéntico al anterior, excepto en que se utiliza una base numérica determinada por el parámetro *base*.
BigInteger(int tamaño, Random r**)**	Genera un número entero largo aleatorio. El número aleatorio abarca el tamaño indicado por el primer parámetro, que se refiere al número de bits que ocupará el número.

Otra forma de crear es mediante el método estático **valueOf** al cual se le puede pasar un entero **long** a partir del cual se devuelve un **BigInteger.** Ejemplo:

```
BigInteger bi=BigInteger.valueOf(2500);
```

métodos

método	uso
BigInteger **abs()**	Obtiene el valor absoluto del número.
BigInteger **add(**BigInteger entero**)**	Devuelve el resultado de sumar el entero actual con el pasado como parámetro
int **bitCount()**	Devuelve el número de bits necesarios para representar el número.
int **compareTo(**BigInteger entero**)**	Compara el entero actual con el utilizado como parámetro. Devuelve -1 si el segundo es mayor que el primero, 0 si son iguales y 1 si el primero era mayor.
BigInteger **divide(**BigInteger entero**)**	Devuelve el resultado de dividir el entero actual entre el parámetro

método	uso
double **doubleValue()**	Obtiene el valor del entero en forma de número **double.**
boolean **equals(**Object o)	Compara el objeto *o* con el entero actual y devuelve **true** si son iguales
double **floatValue()**	Obtiene el valor del entero en forma de número **float.**
BigDecimal **max(**BigDecimal decimal**)**	Devuelve el mayor de los dos números
BigDecimal **min(**BigDecimal decimal**)**	Devuelve el menor de los dos números
BigInteger **mod(**BigInteger entero**)**	Devuelve el resto que se obtiene de dividir el número actual entre el que se pasa como parámetro
BigInteger **multiply(**BigInteger entero**)**	Multiplica los dos números y devuelve el resultado.
BigInteger **negate()**	Devuelve el número multiplicado por menos uno.
BigInteger **probablePrime(**int bits, Random r**)**	Calcula un número primo cuyo tamaño en bits es el indicado y que es generado a partir el objeto aleatorio *r*. La probabilidad de que el número no sea primo es de 2^{-100}
BigInteger **subtract(**BigInteger entero**)**	Resta el entero actual menos el que se recibe como par
BigInteger **toBigInteger()**	Convierte el decimal en **BigInteger**
String **toString()**	Obtiene el número en forma de cadena
String **toString(**int radio**)**	Obtiene el número en forma de cadena usando la base indicada

clase BigDecimal

Se utiliza con más frecuencia, representa números reales de gran precisión. Se usa una escala de forma que el valor de la misma indica la precisión de los decimales. El redondeo se realiza a través de una de estas constantes:

constante	descripción
static int **ROUND_CEILING**	Redondea hacia el infinito positivo
static int **ROUND_DOWN**	Redondea hacia el número 0
static int **ROUND_FLOOR**	Hacia el infinito negativo
static int **ROUND_HALF_DOWN**	Redondea hacia el valor del dígito conexo o cero si coinciden ambos
static int **ROUND_HALF_EVEN**	Redondea hacia el valor del dígito conexo o a un número par si coinciden
static int **ROUND_HALF_UP**	Redondea hacia el valor del dígito conexo o se alejan del cero si coinciden
static int **ROUND_UNNECESSARY**	Se presentan los valores sin redondeos

constante	descripción
static int ROUND_UP	Se redondea alejándose del cero

constructores

constructor	uso
BigDecimal(BigInteger enteroGrande**)**	Crea un **BigDecimal** a partir de un **BigInteger**
BigDecimal(BigInteger enteroGrande, **int** escala**)**	Crea un **BigDecimal** a partir de un **BigInteger** y coloca la escala indicada (la escala determina la precisión de los decimales)
BigDecimal(String texto**)** **throws** NumberFormatException	Crea un objeto para decimales grandes usando el número representado por el texto. En el caso de que el número no sea válido se lanza una excepción **NumberFormatException** El formato del número suele estar en notación científica (2.3e+21)
BigDecimal (double n**)**	Crea un decimal grande a partir del número doble *n*

métodos

método	uso
BigDecimal abs()	Obtiene el valor absoluto del número.
BigDecimal add(BigDecimal decimal**)**	Devuelve el resultado de sumar el decimal actual con el pasado como parámetro
int bitCount()	Devuelve el número de bits necesarios para representar el número.
int compareTo(BigDecimal decimal**)**	Compara el decimal actual con el utilizado como parámetro. Devuelve -1 si el segundo es mayor que el primero, 0 si son iguales y 1 si el primero era mayor.
BigDecimal divide(BigDecimal decimal, **int** redondeo**)**	Devuelve el resultado de dividir el decimal actual entre el *decimal* usado como parámetro. Se le indica el modo de redondeo que es una de las constantes descritas anteriormente.
BigDecimal divide(BigDecimal decimal, **int** escala, **int** redondeo**)**	Idéntica a la anterior, sólo que ahora permite utilizar una escala concreta.
double doubleValue()	Obtiene el valor del decimal en forma de número **double.**
double floatValue()	Obtiene el valor del decimal en forma de número **float.**
boolean equals(Object o**)**	Compara el objeto *o* con el entero actual y devuelve **true** si son iguales

método	uso
BigDecimal **max(**BigDecimal decimal**)**	Devuelve el mayor de los dos números
BigDecimal **min(**BigDecimal decimal**)**	Devuelve el menor de los dos números
BigDecimal **multiply(**BigDecimal decimal**)**	Multiplica los dos números y devuelve el resultado.
BigDecimal **negate()**	Devuelve el número multiplicado por menos uno.
BigDecimal **subtract(**BigDecimal decimal**)**	Resta el entero actual menos el que se recibe como par
int **scale()**	Obtiene la escala del número
void **setScale(**int escala**)**	Modifica la escala del número
String **toString()**	Obtiene el número en forma de cadena

internacionalización. clase Locale

Una de las premisas de Java es la posibilidad de construir aplicaciones que se ejecuten en cualquier idioma y país.

la clase Locale

Toda la cuestión de la portabilidad del código a diversas lenguas, gira en torno a la clase **Locale** (en el paquete **java.util**)**.** Un objeto Locale identifica una configuración regional (código de país y código de idioma). En muchos casos los idiomas soportados están definidos mediante una serie de constantes, que son:

constante	significado
static Locale **CANADA**	**País Canadá**
static Locale **CANADA_FRENCH**	**Idioma francés de Canadá**
static Locale **CHINA**	**País China**
static Locale **CHINESE**	**Idioma chino**
static Locale **ENGLISH**	**Idioma inglés**
static Locale **FRANCE**	**País Francia**
static Locale **FRENCH**	**Idioma francés**
static Locale **GERMAN**	**Idioma alemán**
static Locale **GERMANY**	**País Alemania**
static Locale **ITALIAN**	**Idioma italiano**
static Locale **ITALY**	**País Italia**
static Locale **JAPAN**	**País Japón**
static Locale **JAPANESH**	**Idioma japonés**
static Locale **KOREA**	**País corea del sur**
static Locale **KOREAN**	**Idioma coreano**
static Locale **SIMPLIFIED_CHINESE**	**Idioma chino simplificado**
static Locale **TAIWAN**	**País Taiwán**
static Locale **TRADITIONAL_CHINESE**	**Idioma chino tradicional**

constante	significado
static Locale **UK**	**País Reino Unido**
static Locale **US**	**País Estados Unidos**

La creación de un objeto local para Italia sería:

```
Locale l=Locale.ITALY;
```

No obstante se puede crear cualquier configuración local mediante los constructores que son:

constructor	uso
Locale(String códigoLenguaje**)**	Crea un objeto **Locale** utilizando el código de lenguaje indicado
Locale(String códigoLenguaje, String códigoPaís)	Crea un objeto **Locale** utilizando los códigos de lenguaje y país indicados indicado

Los códigos de países se pueden obtener de la dirección http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt y los códigos de países de: http://ftp.ics.uci.edu/pub/ietf/http/related/iso3166.txt

métodos de Locale

método	descripción
String **getCountry()**	Escribe el código del país del objeto Locale (ES para España)
String **getDisplayCountry()**	Escribe el nombre del país del objeto Locale (España)
String **getLanguage()**	Escribe el código del idioma del objeto Locale (es para Español)
String **getDisplayLanguage()**	Escribe el nombre del idioma (español)
String **getDisplayName()**	Obtiene el texto completo de descripción del objeto local (español España)

métodos estáticos

Permiten obtener información sobre la configuración local de la máquina virtual de Java en ejecución.

método	descripción
static Locale[] **getAvailableLocales()**	Devuelve un array de objetos Locales con la lista completa de objetos locales disponibles en la máquina virtual en ejecución.
static Locale **getDefaultLocale()**	Obtiene el objeto local que se está utilizando en la máquina actual

método	descripción
static String[] getISOCountries()	Obtiene una lista con los códigos de países de dos letras disponibles en la máquina actual
static String[] getISOLanguages	Obtiene una lista con los códigos de idiomas letras disponibles en la máquina actual
static void setDefault(Locale l)	Modifica el objeto Locale por defecto de la máquina actual

formatos numéricos

Ya se vio anteriormente la clase **DateFormat** (ver **clase DateFormat**, página 90). En el mismo paquete de ésta (**java.text**) existen clases dedicadas también al formato de números. Para ello disponemos de **NumberFormat.**

Los objetos **NumberFormat** sirven para formatear números, a fin de mostrarles de forma conveniente. cada objeto **NumberFormat** representa un formato numérico.

creación

Hay tres formas de crear formatos numéricos. Las tres formas se realizan con métodos estáticos (al estilo de **DateFormat**):

método	descripción
static NumberFormat getInstance()	Obtiene un objeto de formato numérico según las preferencias locales actuales
static NumberFormat getInstance(Locale local**)**	Obtiene un objeto de formato numérico basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getCurrencyInstance()	Obtiene un objeto de formato de moneda según las preferencias locales actuales
static NumberFormat geCurrencyInstance(Locale local**)**	Obtiene un objeto de formato de moneda basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getPercentInstance()	Obtiene un objeto de formato porcentaje basado en las preferencias del objeto *local* indicado como parámetro
static NumberFormat getPercentInstance(Locale local**)**	Obtiene un objeto de formato en porcentaje (0,2 se escribiría 20%) basado en las preferencias del objeto *local* indicado como parámetro

USO

El método **format** devuelve una cadena que utiliza el formato del objeto para mostrar el número que recibe como parámetro. Ejemplo:

```
NumberFormat nf=NumberFormat.getCurrencyInstance();
System.out.println(nf.format(1248.32));
//sale 1.248,32 €
```

métodos

Hay métodos que permiten variar el resultado del objeto **NumberFormat:**

método	descripción
boolean getGroupingUsed()	Indica si se está utilizando el separador de miles
int getMinimumFractionDigit()	Devuelve el número mínimo de decimales con el que se formateará a los números
int getMaximumFractionDigit()	Devuelve el número máximo de decimales con el que se formateará a los números
int getMinimumIntegerDigit()	Devuelve el número mínimo de números enteros (los que están ala izquierda del decimal) con el que se formateará a los números. Si hay menos números que el mínimo, se rellenarán los que falta con ceros a la izquierda
int getMaximumIntegerDigit()	Devuelve el número máximo de números enteros con el que se formateará a los números
void setGroupingUsed(boolean uso**)**	Modifica el hecho de que se muestren o no los separadores de miles. Con valor **true** se mostrarán
void setMinimumFractionDigit(int n**)**	Establece el número mínimo de decimales
void setMaximumFractionDigit(int n**)**	Establece el número máximo de decimales
void setMinimumIntegerDigit(int n**)**	Establece el número mínimo de enteros
void setMaximumIntegerDigit(int n**)**	Establece el número máximo de enteros

Propiedades

Las propiedades permiten cargar valores de entorno, esto es valores que se utilizan durante la ejecución de los programas, pero que se almacenan de modo independiente a estos.

Es la clase **Properties (java.util)** la encargada de implementar las propiedades. Esta clase deriva de **Hashtable** con lo que los valores se almacenan en una lista de tipo clave / valor. En el caso de Properties la lista se construye con pares nombre / valor; donde el nombre es el nombre de la propiedad (que se puede agrupar mediante puntos, como los paquetes de Java).

USO

El método para colocar elementos en la lista de propiedades es **put**. Desde la versión 1.2 se utiliza **setProperty** (equivalente a put, pero más coherente con el método de obtención de propiedades **getProperty).** Este método usa dos cadenas, la primera es el nombre de la propiedad y el segundo su valor:

```
Properties prop1=new Properties();
```

```
prop1.setProperty("MiPrograma.maxResult","134");
prop1.setProperty("MiPrograma.minResult","5");
```

Para leer los valores se usa **getProperty,** que devuelve el valor del nombre de propiedad buscado o **null** si no lo encuentra.

```
String mResult=prop1.getProperty("MiPrograma.maxResult");
```

grabar en disco

La ventaja de la clase **Properties** es que las tablas de propiedades se pueden almacenar en discos mediante el método **store**. El método store (que sustituye al obsoleto, **save**) posee dos parámetros, el primero es un objeto de tipo **OutputStream** (ver InputStream/ OutputStream página 93) referido a una corriente de datos de salida en la que se grabará (en forma de texto ASCII) el contenido de la tabla de propiedades. El segundo parámetro es la cabecera de ese archivo. Por ejemplo:

```
prop1.save(System.out,"parámetros de programa");
try{
   File f=new File("d:/propiedades.out");
   FileOutputStream fos=new FileOutputStream(f);
   prop1,save(fos,"parámetros de programa");
}
catch(FileNotFoundException fnf){
   System.out.println("No se encontró el archivo");
}
/*en ambos casos la salida sería:
#parámetros de programa
#Wed Apr 28 00:01:30 CEST 2004
MiPrograma.maxResult=134
MiPrograma.minResult=5
*/
```

A su vez, el método **load** permite leer un archivo de propiedades. Este método devuelve un objeto **Properties** a partir de una corriente **InputStream.**

En ambos métodos (**load** y **store**) hay que capturar las posibles excepciones de tipo **IOException** que se produjeran.

propiedades del sistema

El objeto **System** que se encuentra en **java.lang** representa al propio sistema. Posee un método llamado **getProperty** que permite extraer sus propiedades (el objeto **System** posee una tabla preconfigurada de propiedades). Estas propiedades ocupan el lugar de las variables de entorno de los sistemas operativos.

Las posibles propiedades de sistema que se pueden extraer son:

propiedad	descripción
java.vendor	Fabricante de la máquina Java en ejecución
java.vendor.url	Dirección de la web del fabricante
java.version	Versión de java en ejecución
java.home	Directorio de instalación del kit de java en la máquina actual (inaccesible desde un applet)
os.arch	Arquitectura de Sistema Operativo
os.version	Versión de sistema operativo
os.name	Nombre del sistema operativo
file.separator	Separador de carpetas (\ en Windows)
path.separator	Separador de rutas (; en Windows)
line.separator	Separador de líneas (por ejemplo \n)
user.name	Nombre de usuario (inaccesible desde un applet)
user.home	Ruta a la carpeta de usuario (inaccesible desde un applet)
user.dir	Carpeta actual de trabajo (inaccesible desde un applet)
user.language	Código del lenguaje que utiliza el usuario (por ejemplo "es")
user.country	Código del país del usuario (por ejemplo "ES")
user.timezone	Código de la zona horaria del usuario (por ejemplo *Europe/Paris*)

temporizador

Desde la versión 1.3 de Java hay dos clases que permiten trabajar con temporizadores. Son **java.util.Timer** y **java.util.TimerTask.**

clase TimerTask

Representa una tarea de temporizador; es decir una tarea que se asignará a un determinado temporizador.

Se trata de una clase abstracta, por lo que hay que definir descendientes para poder utilizarla. Cuando se redefine una subclase se debe definir el método abstracto **run**. Este método es el que realiza la operación que luego se asignará a un temporizador.

El método **cancel** permite cancelar la ejecución de la tarea. Si la tarea ya había sido ejecutada, devuelve **false.**

clase Timer

Es la que representa al temporizador. El temporizador se crea usando el constructor por defecto. Mediante el método **schedule** se consigue programar el temporizador. Este

método tiene como primer parámetro un objeto **TimerTask** que tiene que haber programado la acción a realizar cuando se cumpla el tiempo del temporizador.

El segundo parámetro de **schedule** es un objeto **Date** que sirve para indicar cuándo se ejecutará el temporizador.

```
TimerTask tarea=new TimerTask(){
        public void run() {
              System.out.println("Felicidades!!");
        }
};
Timer temp=new Timer();
GregorianCalendar gc=new GregorianCalendar(2007,1,1);
temp.schedule(tarea,gc.getTime());
```

En el ejemplo anterior se escribirá Felicidades!!!, cuando estemos a 1 de enero de 2007 (o en cualquier fecha posterior).

Se puede añadir a **schedule** un tercer parámetro que permite especificar una repetición mediante un número **long** que indica milisegundos:

```
TimerTask tarea=new TimerTask(){
        public void run() {
              System.out.println("Felicidades!!");
        }
};
Timer temp=new Timer();
temp.schedule(tarea,new Date(), 1000);
```

En este caso se ejecuta la tarea en cada segundo. El segundo parámetro en lugar de ser un objeto Date() puede ser también un número de milisegundos desde el momento actual. Así la última línea podía hacerse también con:

```
temp.schedule(tarea, 0, 1000);
```

Finalmente añadir que el método **cancel**, que no tiene argumentos, permite finalizar el temporizador actual.