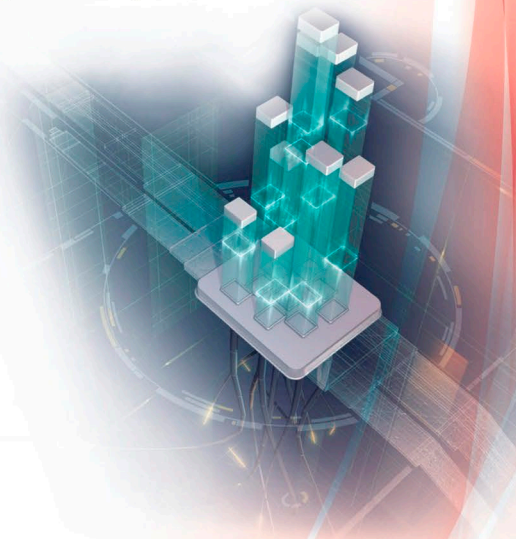
CERTIFICADO DE PROFESIONALIDAD

PROGRAMACIÓN ORIENTADA A OBJETOS

MF0227_3



JUAN CARLOS MORENO PÉREZ

Ra-Ma®

www.ra-ma.es/cp

PROGRAMACIÓN ORIENTADA A OBJETOS

JUAN CARLOS MORENO PÉREZ

Ra-Ma®

La ley prohíbe
Copiar o Imprimir este libro

PROGRAMACIÓN ORIENTADA A OBJETOS
© Juan Carlos Moreno Pérez

ISBN de Edición en Papel: 978-84-9964-509-4

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación y diseño portada: Antonio García Tomé

ISBN:9788-49-964-487-5

E-Book desarrollado en España en Octubre de 2015

A Gaspar,
por todo lo que aprendí de él.

Índice

Introducción

Este libro tiene como finalidad servir de referencia al lector en los certificados de profesionalidad. El objetivo del libro es complementar los contenidos teóricos con la parte práctica de los mismos.

La estructura del libro es tan didáctica como la materia lo permite. Los conceptos presentados en el libro son fáciles de comprender y se acompañan de muchas fotos, ejemplos y explicaciones sencillas. En los contenidos teóricos se ha intentado exponer los conceptos de una forma simplificada, incluyendo siempre los más importantes. Así mismo, se ha procurado que los ejemplos sean lo más sencillos e intuitivos posible.

El libro trata conceptos muy interesantes, como el paradigma de la programación orientada a objetos, el lenguaje Java como un ejemplo de lenguaje orientado a objetos, HTML, CSS, JavaScript, PHP, ingeniería del software, etc.

El lector, para dominar la materia, además de manejar el libro, deberá investigar, realizar ejercicios, documentarse y ampliar conocimientos por sí mismo, puesto que este libro solamente es el empujón en la salida de una carrera ciclista, luego el alumno tendrá que pedalear y recorrer muchos kilómetros solo.

1 Introducción al paradigma orientado a objetos. Clases y objetos

La orientación a objetos es el paradigma de programación más utilizado actualmente. Es difícil programar en un lenguaje que no permita en mayor o menor medida la orientación a objetos. De hecho, muchos lenguajes clásicos se han adaptado a este paradigma para seguir estando de actualidad y no desaparecer.

En este capítulo se introduce al lector en el paradigma de la orientación a objetos y se estudiarán los conceptos de clase y objeto con ejemplos en Java, entre otros. En capítulos posteriores se profundizará más en estos conceptos, pues son la base de la POO.

1.1 EL PARADIGMA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos es un paradigma de programación totalmente diferente al método clásico de programación, el cual utiliza objetos y su comportamiento para resolver problemas y generar programas y aplicaciones informáticas.

Con la programación orientada a objetos (POO) se aumenta la modularidad de los programas y la reutilización de los mismos. Además, la POO se diferencia de la programación clásica porque utiliza técnicas nuevas como el polimorfismo, el encapsulamiento, la herencia, etc.

Generalmente, los lenguajes de última generación permiten la programación orientada a objetos, así como la programación clásica. Por esta razón, puede entenderse la POO como una evolución de la programación clásica (programación estructurada).

1.1.1 CICLO DE DESARROLLO DEL SOFTWARE BAJO EL PARADIGMA DE ORIENTACIÓN A OBJETOS: ANÁLISIS, DISEÑO Y PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos rompe definitivamente con el paradigma de la programación estructurada, surge el concepto de "objeto". Todos los programas se reducen a objetos que tienen una serie de atributos, y, asociados, un comportamiento o procedimientos llamados "métodos". Estos objetos, que son instancias de clases, interaccionarán unos con otros y de esa manera se diseñarán aplicaciones y programas.

Hoy en día es difícil programar sin utilizar programación orientada a objetos (POO). Esta forma de programar es más cercana al modo en que expresaríamos las cosas en la vida real que la que se da en otros tipos de programación clásicos. Los analistas y programadores deben pensar las cosas de una manera distinta para plasmar dichos conceptos en programas en términos de objetos, atributos y métodos.

Figura 1.1. *Ciclo de desarrollo bajo el paradigma de la OO*

El primer paso en el ciclo de desarrollo orientado a objetos es el análisis orientado a objetos del mundo real. Una vez resuelta la fase de análisis se comenzará a diseñar los objetos en detalle para luego pasar a programarlos. Al contrario que en ciclos de desarrollo clásico, donde las fases tenían un comienzo y final claro, las fases definidas en el ciclo de desarrollo orientado a objetos se solapan entre sí. Una vez completadas estas tres fases el producto final es el software terminado, el cual será un compendio de todas las clases de la aplicación.

En Java las aplicaciones son sumamente modulares, el programador creará una serie de programas compuestos por clases. Cada clase está definida en un fichero `.java` independiente que al compilarse generará un fichero `.class`. Para llevar a cabo esta transformación, el programador necesitará de un programa especial (llamado "compilador") que transformará el código Java en unos ficheros `.class` que podrán ser ejecutados en un entorno Java.

EL COMPILADOR DE JAVA

El compilador de Java (javac) viene dentro de un paquete de desarrollo denominado **JDK** (*Java Development Kit* - Kit de desarrollo Java). Si queremos desarrollar programas en Java, como mínimo deberemos instalar un JDK en la máquina donde vayamos a desarrollar

Este proceso se ve en la siguiente figura:

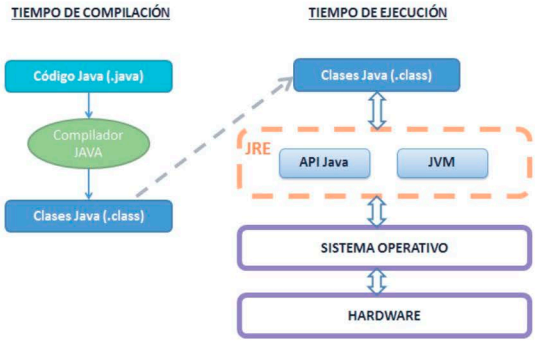


Figura 1.2. *Proceso de compilación y ejecución de un programa Java*

Una vez que el programador compila su programa, el siguiente paso es ejecutarlo. Para hacerlo, el sistema en donde se ejecute deberá tener un **JRE** (*Java Runtime Environment* - Entorno Java de ejecución), el cual contendrá una **JVM** (*Java Virtual Machine* - Máquina virtual Java). A diferencia de otros lenguajes de programación —en los que hay que compilar cada programa para cada sistema operativo—, cuando creas un programa en Java (lo compilas), este funcionará en cualquier sistema siempre y cuando tenga instalada la JVM correspondiente. Cada sistema operativo tendrá una JVM diferente.

¿DISPUESTO A PROGRAMAR?

Entonces descarga e instala el JDK y el JRE en tu máquina.

1.1.2 PROCESO DE CONSTRUCCIÓN DE SOFTWARE: MODULARIDAD. PAQUETES Y LIBRERÍAS

Cuando hablamos de modularidad entendemos por tal la capacidad que tiene una aplicación de ser dividida en varias partes más pequeñas, las cuales tienen que ser independientes unas de otras e incluso deben poder compilarse por separado.

Java —mediante los paquetes— permite modularizar las aplicaciones en varios trozos, los cuales están bien definidos y a la postre pueden servir para reutilizar código.

1.1.2.1 La modularidad en Java: los paquetes

RECUERDA

Librería o paquete son conceptos similares.

Un paquete (o *package*) es un conjunto de clases relacionadas entre sí, las cuales están ordenadas de forma arbitraria. Las clases que forman parte de un paquete no derivan todas ellas de una misma superclase. Por ejemplo, el paquete `java.io` agrupa las clases que permiten a un programa realizar la entrada y salida de información. Un paquete también puede contener a otros paquetes. Con el uso de paquetes se evitan conflictos, como llamar dos clases con el mismo nombre (si existen, estará cada una en paquetes diferentes). Al estar en el mismo paquete, las clases de dichos paquetes tendrán un acceso privilegiado a los miembros de dato y métodos de otras clases del mismo paquete.

RECUERDA

Gracias a los paquetes es posible organizar las clases en grupos. Las clases de un mismo paquete están relacionadas entre sí.

1.1.2.2 La sentencia import

Imaginemos que queremos utilizar una clase contenida en algún paquete, la forma de utilizar dicha clase es generalmente utilizando la sentencia `import`. Podemos importar una clase individual, como por ejemplo:

```
import java.lang.System; // Se importa la clase System
```

O bien podemos importar todas las clases de un paquete:

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

En este caso podremos utilizar todas las clases del paquete `awt` (fíjate en el asterisco). Un ejemplo de esto sería el siguiente:

```
import java.awt.*;
....
Frame fr = new Frame( "Panel ejemplo" );
```

También es posible utilizar la clase sin utilizar la sentencia `import`, lo cual muchas veces no es aconsejable puesto que el código se hace muy voluminoso:

```
java.awt.Frame fr = new java.awt.Frame( "Panel ejemplo" );
```

Generalmente, cuando se van a crear varios objetos de una o varias clases de un paquete, se desaconseja esta última opción.

1.1.2.3 Localización de librerías

Para encontrar una clase u otro recurso, Java necesita dos cosas:

- El nombre del paquete.
- Las rutas donde están situados los paquetes y las clases (`path` de búsqueda más conocido como `CLASSPATH`).

El **`CLASSPATH`** sirve para localizar clases creadas por el usuario o terceras personas que no son parte de la plataforma Java. Establecer el valor de esta variable es necesario cuando se va a utilizar una clase que no está en el mismo directorio (o subdirectorios) de la clase donde se está trabajando, o no está en ningún lugar definido por el mecanismo de extensiones.

Una alternativa a establecer el valor de la variable de entorno `CLASSPATH` es utilizar las opciones `-cp` o `-classpath` (es lo mismo) al ejecutar Java (java, javac, javah o jdb).

Imaginemos que tenemos un paquete `utilidades.proyecto` y dentro de él una clase que se llama `programa.class`. Esta clase se encuentra en la siguiente ruta:

```
/java/Misclases/utilidades/proyecto
```

Para ejecutar la clase deberíamos ejecutar el siguiente comando:

```
% java -classpath /java/Misclases utilidades.proyecto.programa
```

Otra opción es establecer la variable de entorno `CLASSPATH`:

```
CLASSPATH = /java/Misclases:path2:path3:...
export CLASSPATH
```

Y luego ejecutar el siguiente comando:

```
% java utilidades.proyecto.programa
```

Java se encargaría de buscar la clase en el directorio especificado por la variable `CLASSPATH`.

IMPORTANTE

Los valores de la variable de entorno `CLASSPATH` están separados por dos puntos ":" en Linux/Unix y por punto y coma ";" en sistemas Windows.

1.1.3 INTRODUCCIÓN AL CONCEPTO DE OBJETO

Los programas realizados mediante el paradigma de la POO solamente tienen objetos. Todos los objetos pertenecen a una clase; por ejemplo, mi loro *Felipe* pertenecería a la clase *pájaro*. Todos los objetos de la clase *pájaro* se identificarán, entre otros atributos, con un nombre (en este caso, *Felipe*), un color de plumaje (verde), una edad (en este caso, dos años) y si son domésticos o no (*Felipe* sí es doméstico: lo tengo ahora en mi hombro).

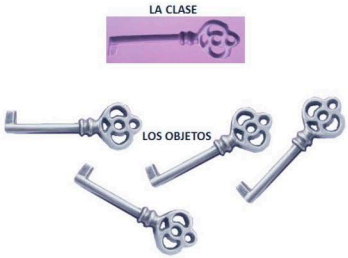


Figura 1.3. *Símil visual de lo que serían una clase y varios objetos de dicha clase*

RECUERDA

En un símil con la costura, las clases serían los patrones y los objetos las prendas.

Clases. Las clases son los moldes de los cuales se generan los objetos. Los objetos se instancian y se generan, con lo cual "instancia" y "objeto" son sinónimos. Por ejemplo, *Felipe* será un objeto concreto de la clase *pájaro*.

RECUERDA

Cuando se escribe un programa o aplicación OO, lo que se hace es definir las clases de objetos dotándolas de estado y comportamiento; y cuando se ejecute el programa se crearán los objetos, ya sea estática o dinámicamente.

Cuando se programa, las clases se escriben en ficheros ASCII con el mismo nombre que la clase y extensión `.java`.

Las clases tienen una estructura parecida a la siguiente:

```
[algo_1] class nombre_de_la_clase [algo_2] {
          [Atributos]
          [Métodos]
}
```

Como puedes observar, se han etiquetado con corchetes (`[ ]`) los elementos opcionales de la clase. También hay dos elementos opcionales etiquetados como `algo_1` y `algo_2` que contendrán palabras reservadas que se estudiarán en los siguientes capítulos. Es muy común ver definiciones de clases como `public class nombre_de_la_clase`. La palabra reservada `public` indica que la clase puede ser accedida por cualquier clase que necesite de su utilización. Entre los corchetes, dentro de la clase, encontraremos los atributos (una clase puede tener cero o muchos atributos) y los métodos (una clase puede tener cero o muchos métodos). Los métodos –para la gente que haya programado en cualquier lenguaje de programación– son los llamados "procedimientos" o "funciones".

RECUERDA

En Java, el nombre de la clase y el del fichero que la contiene deberá ser el mismo.

- **Objetos**. Un objeto tiene las siguientes características:
  - **Identidad**. Cada objeto es único y diferente de otro objeto. Mi loro *Felipe* es diferente a otros loros, aunque estos sean verdes, tengan dos años y sean también domésticos.
  - **Estado**. El estado serán los valores de los atributos del objeto, en el caso de los objetos de la clase *pájaro* serían *nombre*, *color*, *edad*, *doméstico*, etc.
  - **Comportamiento**. El comportamiento serían los métodos o procedimientos que realiza dicho objeto. Dependiendo del tipo o clase de objeto, estos realizarán unas operaciones u otras (cantar, volar, hablar, etc.).

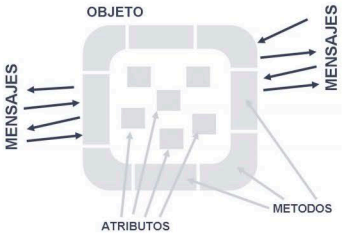


Figura 1.4. *Estructura de un objeto*

- **Mensajes**. Como se dijo antes, los programas o aplicaciones orientadas a objetos están compuestos por objetos, los cuales interactúan unos con otros a través del paso de mensajes. Cuando un objeto recibe un mensaje, lo que hace es ejecutar el método asociado.

- **Métodos**. Los métodos son los procedimientos que ejecuta el objeto cuando recibe un mensaje vinculado a ese método concreto. En ocasiones este método envía mensajes a otros objetos, solicitando acciones o información.

RECUERDA

En un programa OO primeramente se crean los objetos y entre ellos se envían mensajes, la información se procesa para luego destruirse y liberar la memoria que estaban ocupando.

Los objetos del mundo real tienen dos características principales:

- Estado.
- Comportamiento.

Los loros, por ejemplo, tienen un estado que puede ser el color del plumaje, el nombre, si hablan o no, etc. Y también un comportamiento (hablar, piar, comer, etc.).

Algunos objetos son más complejos que otros; por ejemplo, mi consola es mucho más compleja que mi linterna de espeleología. Mi linterna tiene solo dos estados (encendida y apagada) y la interfaz es sumamente sencilla (apagar y encender).

Al programar una aplicación en Java deberemos modelar estos estados y comportamientos en clases y objetos.

La programación orientada a objetos proporciona los siguientes beneficios:

- **Modularidad**. El código fuente de un objeto puede mantenerse y reescribirse sin que ello implique la reprogramación del código de otros objetos de la aplicación.

- **Reutilización de código**. Es muy sencillo utilizar clases y objetos de terceras personas. La ventaja de esto es que no tenemos que conocer los detalles de su implementación interna, sino solamente su interfaz.

- **Facilidad de testeo y reprogramación**. Si tenemos un objeto que está dando problemas en una aplicación, no tenemos que reescribir el código de toda la aplicación: basta con reemplazar el objeto por otro similar, o bien reprogramarlo.

- **Ocultación de información**. En la POO se ocultan los detalles de implementación y lo que prima es la interfaz.

1.2 CLASES Y OBJETOS

En la programación orientada a objetos, las clases permiten a los programadores abstraer el problema que se busca resolver ocultando los datos y la manera en la que estos se manejan para llegar a la solución (se oculta la implementación). En los siguientes apartados se estudiarán en profundidad los conceptos de clase y objeto.

1.2.1 LAS CLASES. PROPIEDADES Y MÉTODOS

En un programa orientado a objetos es impensable que desde el mismo programa se acceda directamente a las variables internas de una clase si no es a través de métodos **getters** y **setters** (por ejemplo `getEdad()` o `setEdad()`).

IMPORTANTE

La abstracción es importante en el análisis y diseño de aplicaciones orientadas a objetos. La finalidad del A&D es crear un conjunto de clases que resuelvan el problema que se está abordando.

Por lo tanto, en la definición de nuestras clases deberemos cuidar lo siguiente:

- No se deberá tener acceso directo a la estructura interna de las clases. El acceso a los atributos será a través de *getters* y *setters*.
- En el supuesto de que haya que modificar el código sin modificar la interfaz con otras clases o programas, esto debería poder hacerse sin que tenga ninguna repercusión con otras clases o programas. Se busca que las clases tengan un alto grado de cohesión (independencia).

En Java hay varios niveles de acceso a los miembros de una clase:

- **`public`** (acceso público).
- **`protected`** (acceso protegido).
- **`private`** (acceso privado).
- **`no especificado`** (acceso en su paquete).

Cuando especificamos el nivel de acceso a un atributo o método de una clase, lo que estamos especificando es el nivel de accesibilidad que va a tener ese atributo o método, que puede ir desde el acceso más restrictivo (`private`) al menos restrictivo (`public`).

RECUERDA

Una subclase es una clase que hereda ciertas características de la clase padre, aunque puede añadir algunas propias. Las subclases se estudiarán en profundidad más adelante.

Dependiendo de la finalidad de la clase, utilizaremos un tipo de acceso u otro.

- **Acceso público** (`public`). Un miembro público puede ser accedido desde cualquier otra clase o subclase que necesite utilizarlo. Una interfaz de una clase estará compuesta por todos los miembros públicos de la misma.
- **Acceso privado** (`private`). Un miembro privado puede ser accedido solamente desde los métodos internos de su propia clase. Otro acceso será denegado.
- **Acceso protegido** (`protected`). El acceso a estos miembros es igual que el acceso privado. No obstante, para las subclases o clases del mismo paquete (*package*) a la que pertenece la clase, se considerarán estos miembros como públicos.
- **Acceso no especificado** (`paquete`). Los miembros no etiquetados podrán ser accedidos por cualquier clase perteneciente al mismo paquete.

CONSEJO

Para un mayor control de acceso se recomienda etiquetar los miembros de una clase como `public`, `private` y `protected`.

A modo de resumen se especificarán los niveles de acceso vistos anteriormente en la siguiente tabla:

Modificador de acceso	`public`	`protected`	`private`	Sin especificar (acceso paquete)
¿El método o atributo es accesible desde la propia clase?	SÍ	SÍ	SÍ	SÍ
¿El método o atributo es accesible desde otras clases en el mismo paquete?	SÍ	SÍ	NO	SÍ
¿El método o atributo es accesible desde una subclase en el mismo paquete?	SÍ	SÍ	NO	SÍ
¿El método o atributo es accesible desde subclases en otros paquetes?	SÍ	(*)	NO	NO
¿El método o atributo es accesible desde otras clases en otros paquetes?	SÍ	NO	NO	NO

(*) Este caso no se suele dar con frecuencia. Se podría acceder al atributo o método desde objetos de la subclase, pero no así por objetos de la superclase.

Tabla 1.1. *Modificadores de acceso en Java*

Cuando creamos una clase en Java es posible definir la relación que esa clase tiene con otras clases o la relación que tendrá esa clase con las clases de su mismo paquete.

CONSEJO

Una clase definida como pública puede ser utilizada por las clases de su paquete y otros paquetes mientras que una clase no definida como pública solamente podrá ser utilizada por las clases de su propio paquete.

Clase pública	Clase NO definida como pública
`public class miClase { ..... }`	`class miClase { ..... }`
Puede ser utilizada por cualquier clase.	Puede ser utilizada SOLO por clases de su propio paquete.

RECUERDA

Datos, propiedades o atributos son sinónimos y referencian a las variables de una clase.

En una clase se agrupan datos (variables) y métodos (funciones). Todas las variables o funciones creadas en Java deben pertenecer a una clase, con lo cual no existen variables o funciones globales como en otros lenguajes de programación.

RECUERDA

En una clase Java los atributos pueden ser tipos primitivos (`char`, `int`, `boolean`, etc.) o bien pueden ser objetos de otra clase. Por ejemplo, un objeto de la clase `coche` puede tener un objeto de la clase `motor`.

En la siguiente clase se puede observar perfectamente los atributos de la clase y los métodos:

```
class pajaro
{   //*** atributos o propiedades ****
    private char color; //propiedad o atributo color
    private int edad; //propiedad o atributo edad
    //*** métodos de la clase ****
    public void setedad(int e){edad = e;}
    public void printedad(){System.out.println(edad);}
    public void setcolor(char c){color=c;}
    public void printcolor(){
           switch(color){
```

```
            //Los pájaros son verdes, amarillos, grises, negros o blancos
            //No existen pájaros de otros colores
                  case 'v': System.out.println("verde");break;
                  case 'a': System.out.println("amarillo");break;
                  case 'g': System.out.println("gris");break;
                  case 'n': System.out.println("negro");break;
                  case 'b': System.out.println("blanco");break;
                  default: System.out.println("color no establecido");
            }
    }
}

class test
{
    public static void main(String[] args) {
            pajaro p;
            p=new pajaro();
            p.setedad(5);
            p.printedad();
    }
}
```

Como se puede ver en el código anterior existen dos clases diferentes (`pájaro` y `test`), las cuales residirán en dos ficheros diferentes (`pajaro.java` y `test.java`). En la clase `test` se crea un objeto de la clase `pájaro` y se llama a los métodos para actualizar la edad y mostrarla por pantalla. En ningún momento la clase `test` puede acceder a los métodos o atributos `private` de la clase `pájaro` (esto es gracias a la **abstracción**); ni falta que le hace, porque lo único que debe conocer la clase `test` son los métodos públicos para utilizar la clase.

1.2.2 OBJETOS: ESTADO, COMPORTAMIENTO E IDENTIDAD

Los métodos pueden permitir que se los llame especificando una serie de valores. A estos valores se les denomina "parámetros". Los parámetros pueden tener un tipo básico (`char`, `int`, `boolean`, etc.) o bien ser un objeto. Además, los métodos (salvo el constructor) pueden retornar un valor o no (en ese caso se pone `void`).

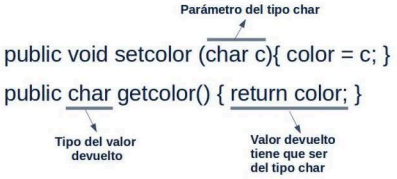


Figura 1.5. *Parámetros y valores devueltos*

En la figura anterior se pueden ver dos métodos para la clase `pájaro`: `setcolor`, la cual admite un parámetro, y `getcolor`, la cual devuelve un valor de tipo `char`.

En Java existen unos métodos especiales que son los constructores y destructores del objeto. Estos métodos son opcionales, es decir, no es obligatorio programarlos salvo que se necesiten.

- El **constructor** del objeto es un procedimiento llamado automáticamente cuando se crea un objeto de esa clase. Si el programador no los declara, Java generará uno por defecto. La función del constructor es inicializar el objeto.
- El **destructor**, por el contrario, se ejecutará automáticamente siempre que se destruye un objeto de dicha clase. Los destructores, generalmente, se utilizan para liberar recursos y cerrar flujos abiertos (realiza una limpieza final). Los destructores no reciben parámetros, al contrario que los constructores, la sobrecarga no está permitida. En C++, el destructor se denomina igual que el constructor, nada más que se le coloca delante el símbolo ~. En Java, la destrucción de objetos sigue otra filosofía distinta a la de otros lenguajes de programación. El sistema de destrucción de objetos ya se verá más adelante en profundidad.

RECUERDA

En Java NO hay destructores como en C++.

En el siguiente código se verán los constructores para la clase `pájaro`:

```
class pajaro
{   //*** atributos o propiedades ****
    private char color; //propiedad o atributo color
    private int edad; //propiedad o atributo edad
    //*** métodos de la clase ****
    pajaro(){color = 'v'; edad = 0;} //constructor de la clase pájaro
    pajaro(char c, int e){color = c; edad = e;} // constructor de la clase pájaro
    /* Aquí irán los demás métodos de la clase */
    public static void main(String[] args) { //método main
           pajaro p1,p2;
           p1=new pajaro();
           p2=new pajaro('a',3);
    }
}
```

Como se puede ver, el constructor de la clase `pájaro` está sobrecargado (existe más de un constructor en la clase). Eso quiere decir que es posible crear objetos de la clase `pájaro` de distintas formas.

1.2.3 OBJETOS COMO INSTANCIAS DE CLASE. LA INSTANCIA ACTUAL THIS

Como ya se ha dicho, los programas realizados mediante el paradigma de la POO solamente tienen objetos, y todos y cada uno de estos objetos pertenecen a una clase concreta. Además Java mantiene un puntero a la instancia actual, al que llama **`this`**.

Java, al igual que C++, proporciona una referencia al objeto con el que se está trabajando. La referencia `this` no es ni más ni menos que el objeto que está ejecutando el método. En los ejemplos que hemos estado utilizando, en muchas ocasiones se obviaba esta referencia puesto que se sobreentiende que el objeto está invocando al método. En algunas ocasiones nos va a servir para resolver ambigüedades o para devolver referencias al propio objeto. En el siguiente ejemplo se ve claramente el uso del `this`.

OBSERVA

En el siguiente código vas a poder apreciar que la referencia `this` en ocasiones se puede omitir. Observa también cómo se devuelve una referencia al propio objeto en los métodos `incrementarAncho()` e `incrementarAlto()`.

```
class rectangulo
{
    private int ancho = 0;
    private int alto = 0;
    rectangulo(int an, int al){
           ancho = an; //se puede omitir el this
           this.alto = al;
    }
    public int getAncho(){return this.ancho;}
    public int getAlto(){return alto;} //se puede omitir el this
    public rectangulo incrementarAncho(){
           ancho++; //se puede omitir el this
           return this;
    }
    public rectangulo incrementarAlto(){
           this.alto++;
           return this;
    }
}
```

1.2.4 USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

En este apartado se va a ver cuándo un método o atributo debe ser estático y cuándo no. Cuando un método o atributo se define como **`static`** quiere decir que se va a crear para esa clase solo una instancia de ese método o atributo. En el siguiente caso se ve cómo se ha creado un atributo `numpajaros` que contará el número de pájaros que se van generando. Si ese atributo no fuera estático sería imposible contar los pájaros, puesto que en cada instancia del objeto se crearía una variable `numpajaros`. De la misma manera, los métodos `nuevopajaro()`, `muestrapajaros()` o el `método main()` tiene sentido que sean estáticos.

```
class pajaro
{   //*** atributos o propiedades ****
    private static int numpajaros=0;
    private char color; //propiedad o atributo color
```

```
    private int edad; //propiedad o atributo edad
    //*** métodos de la clase ****
    static void nuevopajaro(){numpajaros++;};
    pajaro(){color = 'v'; edad = 0; nuevopajaro();}
    pajaro(char c, int e){color = c; edad = e; nuevopajaro();}
    static void muestrapajaros(){System.out.println(numpajaros);};
    public static void main(String[] args) {
           pajaro p1,p2;
           p1=new pajaro();
           p2=new pajaro('a',3);
           p1.muestrapajaros();
           p2.muestrapajaros();
    }
}
```

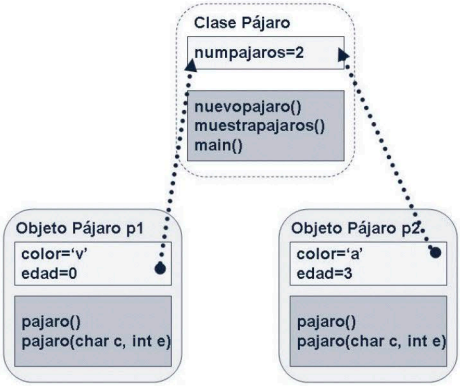


Figura 1.6. *Clase pájaro y objetos de dicha clase*

Como se puede ver en la figura anterior, el atributo numpajaros y los métodos `nuevopajaro()`, `muestrapajaros()` y `main()` se comparten por todos los objetos creados de la clase `pájaro`.

1.2.5 LOS PAQUETES

Los conceptos de paquete, librería y `CLASSPATH` se han visto en apartados anteriores. Sobre los paquetes hay que tener en cuenta las siguientes ideas:

- Un paquete es un conjunto de clases relacionadas entre sí.
- Un paquete puede contener a su vez subpaquetes.

- Java mantiene su biblioteca de clases en una estructura jerárquica.
- Cuando nos referimos a una clase de un paquete (salvo que se haya importado el paquete) hay que referirse a la misma especificando el paquete (y subpaquete si es necesario) al que pertenece (por ejemplo: `java.io.File`).
- Los paquetes permiten reducir los conflictos con los nombres puesto que dos clases que se llaman igual, si pertenecen a paquetes distintos, no deberían dar problemas.
- Los paquetes permiten proteger ciertas clases no públicas al acceso desde fuera del mismo.

Para la **creación de un paquete** muy sencillo vamos a seguir los siguientes pasos:

El objetivo es crear un paquete con dos clases y llamar a dichas clases desde un programa aparte.

1 Lo primero que hay que hacer es crear en el directorio donde estamos compilando los programas un subdirectorio con nombre; por ejemplo, `Utilidades`. En este subdirectorio vamos a tener varios paquetes (serán subpaquetes del paquete `utilidades`). El primero de ellos se va a llamar `educación` y vamos a tener dentro de él dos clases ya compiladas llamadas `saludar` y `despedirse` (`saludar.class` y `despedirse.class`).

Figura 1.7. *Estructura de directorios de los paquetes que se van a crear*

Cada subpaquete estará situado en un subdirectorio aparte del subdirectorio `Utilidades`. Como se puede observar en la imagen anterior, se está utilizando el compilador Geany. En este directorio (`Geany`) se guardan los programas y las clases, y es aquí donde crearemos estos subdirectorios.

2 Las clases que se crearán son las siguientes:

```
package Utilidades.educacion;
import java.io.*;
public class saludar{
    public void saludo(){
          System.out.println("Hola");
    }
}
//** Fin código****
//** Inicio código****
package Utilidades.educacion;
import java.io.*;
```

```
public class despedirse{
    public void despedida(){
          System.out.println("Adios");
    }
}
```

```
package Utilidades.educacion;
```

Nótese que, con la sentencia anterior, ambas clases indican que pertenecen al paquete `educación`.

3 Una vez que he hecho eso, el siguiente paso será importar el paquete con la sentencia `import`.

```
import Utilidades.educacion.*;
public class test {
public static void main(String[] args) {
                        saludar s=new saludar();
                        despedirse d=new despedirse();
                        s.saludo();
                        d.despedida();
}
}
```

En el programa anterior se crearán dos objetos, uno de cada clase (`saludar` y `despedirse`) y se hace una llamada a un método de cada clase para verificar que el paquete funciona correctamente. Si se han realizado estos pasos correctamente, la compilación no debería dar ningún error.

1.2.6 CONCEPTO DE "PROGRAMA" EN EL PARADIGMA ORIENTADO A OBJETOS

DEFINICIÓN DE PROGRAMA

Un programa es una serie de órdenes o instrucciones ordenadas con una finalidad concreta que realizan una función determinada.

Todos estamos familiarizados con la ejecución de programas (editores de textos, navegadores, juegos, reproductores de música o películas, etc.). Por regla general, cuando queremos ejecutar un programa se lo indicamos al sistema haciendo doble clic sobre él, e incluso algunos usuarios más avanzados ejecutan comandos desde un intérprete de comandos o consola. Si alguna vez has tenido la curiosidad de abrir un programa con un bloc de notas o editor de texto te habrás dado cuenta de que aparece algo horrible en el editor: una serie de símbolos ininteligibles (por los humanos). Eso es porque los programas están en binario, que es el lenguaje que entienden las máquinas. Entonces te preguntarás: si al final de este libro seré capaz de escribir programas, ¿podré entender esos códigos? La respuesta es no. En este libro vamos a aprender un lenguaje de programación para escribir programas de manera entendible por los humanos, que luego traduciremos al lenguaje máquina entendible por los ordenadores mediante otros programas llamados "intérpretes" o "compiladores".

En la siguiente figura se verá todo esto de modo más gráfico:

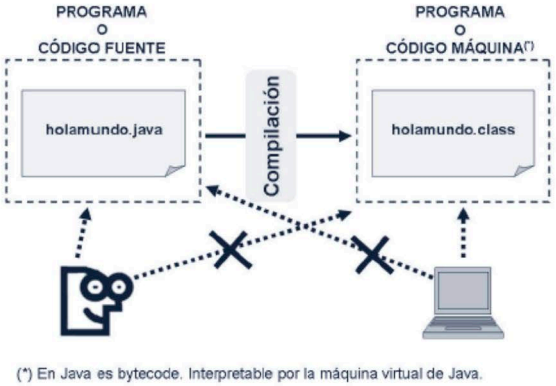


Figura 1.8. *Proceso de compilación en Java*

Como se puede observar, el código fuente es el que escribe el programador, que luego lo compila a código máquina. Compilar equivale a transformar el programa inteligible por el programador al programa inteligible por la máquina. El código fuente o programa fuente está escrito en un lenguaje de programación y el compilador es un programa que se encarga de transformar el código fuente en código máquina.

Los compiladores son programas específicos para un lenguaje de programación, los cuales transforman el programa fuente en un programa directa o indirectamente ejecutable por la máquina destino. No es posible compilar un programa escrito en lenguaje Java con un compilador de C porque este no lo entendería.

El lenguaje máquina que genera Java es un lenguaje intermedio interpretable por una máquina virtual instalada en el ordenador donde se va a ejecutar. Una máquina virtual es una máquina ficticia que traduce las instrucciones máquina ficticias a instrucciones para la máquina real. La ventaja de la misma es que los programas se pueden ejecutar en cualquier tipo de hardware siempre y cuando tenga instalada la máquina virtual correspondiente (la JVM). Los programas no van a cambiar, lo que cambiará es la máquina virtual dependiendo del hardware (no será igual la máquina virtual de un *smartphone* que la de un PC).

LOS COMPILADORES E INTÉRPRETES

A diferencia de los compiladores, los intérpretes leen línea a línea el código fuente y lo ejecutan. Este proceso es muy lento y requiere tener cargado en memoria el intérprete. La ventaja de los intérpretes es que la depuración y corrección de errores del programa es mucho más sencilla que con los compiladores.

Java es uno de los lenguajes más utilizados en la actualidad. Es un lenguaje de propósito general y su éxito radica en que es el lenguaje de Internet. *Applets*, *servlets*, páginas JSP o JavaScript utilizan Java como lenguaje de programación.

El éxito de Java reside en que es un lenguaje multiplataforma. Java utiliza una máquina virtual en el sistema destino y por lo tanto no hace falta recompilar de nuevo las aplicaciones para cada sistema operativo. Java, por lo tanto, es un lenguaje interpretado que para mayor eficiencia utiliza un código intermedio (*bytecode*). Este código intermedio o *bytecode* es independiente de la arquitectura y por lo tanto puede ser ejecutado en cualquier sistema.

CONCEPTO DE PROGRAMA EN LA POO

Un programa en POO se organiza en clases. Dichas clases tienen datos más operaciones que se hacen sobre dichos datos (métodos). Cuando se ejecuta un programa se crearán varios **objetos**, cada uno de los cuales será instancia de una clase, y se pasarán **mensajes** entre ellos, realizando de esta manera las tareas que demanda el usuario.

1.3 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:
- a) Con la programación orientada a objetos (POO) se aumenta la modularidad.
- b) Con Java no se puede realizar programación clásica (programación estructurada).
- c) El constructor del objeto es un procedimiento llamado automáticamente cuando se crea un objeto de esa clase.
- d) El `CLASSPATH` sirve para localizar clases creadas por el usuario o terceras personas.

2 ¿Cuál de las siguientes afirmaciones es falsa?:
- a) Un paquete es un conjunto de clases relacionadas entre sí.
- b) Un método definido como `protected` puede ser accedido desde la misma clase pero no desde subclases en el mismo paquete.
- c) En Java no existen variables o funciones globales como en otros lenguajes de programación.
- d) El compilador de Java viene dentro de un paquete de desarrollo denominado JDK.

3 ¿Cuál de las siguientes afirmaciones es falsa?:
- a) Las clases son los moldes de los cuales se generan los objetos.
- b) Las clases se escriben en un lenguaje llamado *bytecode*.
- c) En Java NO hay destructores como en C++.
- d) El polimorfismo, el encapsulamiento y la herencia son técnicas nuevas de la POO.

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El estado de un objeto variará dependiendo de los valores de los atributos del objeto.
b) Una clase definida como `pública` puede ser utilizada por las clases de su paquete y otros paquetes.
c) El compilador de Java viene dentro de un paquete de desarrollo denominado JRE.
d) Un *package* es un conjunto de clases relacionadas entre sí.

5 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Los métodos son los procedimientos que ejecuta el objeto cuando recibe un mensaje.
b) El destructor se ejecutará automáticamente siempre que se destruye un objeto de dicha clase.
c) Un método no definido como `private` no es accesible desde otras clases en el mismo paquete.
d) Java proporciona una referencia al objeto con el que se está trabajando que se llama `this`.

6 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un paquete puede contener a su vez subpaquetes.
b) Para realizar programas en Java, necesitamos un JDK y un JRE en la máquina de desarrollo.
c) Cuando un método o atributo se define como `static` quiere decir que se va a crear para esa clase solo una instancia de ese método o atributo.
d) Un compilador puede compilar programas realizados en varios lenguajes de programación.

2 Técnicas de programación estructurada

La programación estructurada surgió en la década de los 60 y es un paradigma de programación en el que se evita a toda costa la instrucción de salto incondicional `GOTO`, los programas estructurados utilizan subrutinas y tres estructuras básicas como son la **secuencial**, la de **selección** (`if` y `switch`) y la de **iteración** (`for` y `while`). El objetivo de la programación estructurada es realizar programas con más claridad, con más calidad, fáciles de mantener y con menos errores.

Otra característica de la programación estructurada es que el esfuerzo en las pruebas y depuración de los programas es menor, dado que la estructura de los mismos es más sencilla. También, a la hora de mantener los programas, las modificaciones son más rápidas, dado que son más claros e intuitivos. Además, los programadores incrementan su rendimiento por las razones anteriormente citadas.

LOS PROGRAMAS EN JAVA

Los programas o aplicaciones en Java se componen de varios ficheros `.class`, que son ficheros en **bytecode** que contienen las clases del programa. Estos ficheros no tienen por qué estar situados en un directorio concreto, sino que pueden estar distribuidos en varios discos o incluso en varias máquinas.
La aplicación se ejecuta desde el método o procedimiento principal `main()` situado en una clase o fichero principal.

2.1 ELEMENTOS BÁSICOS: CONSTANTES, VARIABLES, OPERADORES Y EXPRESIONES

2.1.1 TIPOS DE DATOS SIMPLES

Los tipos de datos se utilizan generalmente al declarar variables y son necesarios para que el intérprete o compilador conozca de antemano el tipo de información que va a contener una variable. Los tipos de datos primitivos en Java son los siguientes:

Tipo de datos	Información representada	Rango	Descripción
`byte`	Datos enteros	-128 ←→ +127	Se utilizan 8 bits (1 byte) para almacenar el dato.
`short`	Datos enteros	-32768 ←→ +32767	Dato de 16 bits de longitud (independientemente de la plataforma).
`int`	Datos enteros	-2147483648 ←→ +2147483647	Dato de 32 bits de longitud (independientemente de la plataforma).
`long`	Datos enteros	-9223372036854775808 ←→ +9223372036854775807	Dato de 64 bits de longitud (independientemente de la plataforma).

`char`	Datos enteros y caracteres	0 ←→ 65535	Este rango es para representar números en unicode, los ASCII se representan con los valores del 0 al 127. ASCII es un subconjunto del juego de caracteres Unicode.
`float`	Datos en coma flotante de 32 bits	Precisión aproximada de 7 dígitos	Dato en coma flotante de 32 bits en formato IEEE 754 (1 bit de signo, 8 para el exponente y 24 para la mantisa).
`double`	Datos en coma flotante de 64 bits	Precisión aproximada de 16 dígitos	Dato en coma flotante de 64 bits en formato IEEE 754 (1 bit de signo, 11 para el exponente y 52 para la mantisa).
`boolean`	Valores booleanos	true/false	Utilizado para evaluar si el resultado de una expresión booleana es verdadero (`true`) o falso (`false`).

Tabla 2.1. *Tipos de datos simples*

ACTIVIDAD PROPUESTA

Se propone al lector que investigue y recopile información sobre el juego de caracteres Unicode y ASCII, con especial detenimiento en este último.

A continuación, se muestran ejemplos de utilización de tipos de datos en la declaración de variables:

Tipo de dato	Código
byte	byte a;
short	short b, c=3;
int	int d = -30; int e = 0xC125;
long	long b=434123 ; long b=5L ; /* la L en este caso indica Long*/
char	char car1='c'; char car2=99; /*car1 y car2 son lo mismo porque el 99 en ASCII es la 'c' */
float	float pi=3.1416; float pi=3.1416F; /* la F en este caso indica Float*/ float medio=1/2F; /*0.5*/
double	double millón=1e6; /* $1x10^6$ */ double medio=1/2D; /*0.5 la D en este caso indica Double*/
boolean	boolean adivinanza=true;

Tabla 2.2. *Utilización de tipos de datos*

2.1.2 CONSTANTES Y LITERALES

2.1.2.1 Las constantes

CUESTIÓN DE ESTILO

Las constantes se declaran en mayúscula; las variables, en minúscula (esto se realiza como norma de estilo).

Una constante es un dato invariable, siempre es el mismo. Las constantes se declaran siguiendo el siguiente formato:

```
final [static] <tipo de datos> <nombre de la constante> = <valor>;
```

Donde el calificador final identificará que es una constante, la palabra **static** si se declara implicará que solo existirá una copia de dicha constante en el programa, aunque se declare varias veces, el tipo de datos de la constante seguido del nombre y por último el valor que toma.

```
final static double PI=3.141592;
```

RECUERDA

Las constantes se utilizan en datos que nunca varían (IVA, PI, etc.). Utilizando constantes y no variables nos aseguramos de que su valor no va a poder ser modificado nunca. También utilizar constantes permite centralizar el valor de un dato en una sola línea de código (si se quiere cambiar el valor del IVA, se hará solamente en una línea en vez de si se utilizase el literal 18 en muchas partes del programa).

2.1.2.2 Los literales

Un literal puede ser una expresión:

- De tipo de **dato simple**.
- **Nula** o de valor `null`.
- Un **string** o cadena de caracteres (por ejemplo `"Hola Mundo"`).

Ejemplos de literales en Java pueden ser `'a'`, `322`, `3.1416`, `"pi"` o `"programación en Java"`.

2.1.3 VARIABLES

Una variable no es ni más ni menos que una zona de memoria donde se puede almacenar información del tipo que desee el programador.

LAS PALABRAS CLAVE

Las palabras clave son las órdenes del lenguaje de programación. El compilador espera esos identificadores para comprender el programa, compilarlo y poder ejecutarlo. Por lo tanto queda PROHIBIDO utilizar palabras clave (como `boolean`, `double`, `long`, `if`, `private`, etc.) utilizadas por el propio Java para nombrar variables dentro de un programa. Tampoco se pueden utilizar caracteres especiales para nombrar variables (como +, -, /, etc.).

```
class suma
{
    static int n1=50; // variable miembro de la clase
    public static void main(String [] args)
    {
    int n2=30, suma=0; // variables locales
    suma=n1+n2;
    System.out.println("LA SUMA ES: " + suma);
    }
}
```

Como puede verse en el ejemplo anterior, las variables se declaran dentro de un bloque (por bloque se entiende el contenido entre las llaves `{}`) y son accesibles solo dentro de ese bloque.

Las variables declaradas en el bloque de la clase como `n1` se consideran miembros de la clase, mientras que las variables `n2` y `suma` pertenecen al método `main` y solo pueden ser utilizadas en el mismo. Las variables declaradas en el bloque de código de un método son variables que se crean cuando el bloque se declara, y se destruyen cuando finaliza la ejecución de dicho bloque.

Las variables **miembros** de una clase **se inicializan por defecto** (las numéricas con `0`, los caracteres con `'\0'` y las referencias a objetos y cadenas con `null`), mientras que las variables **locales no lo hacen**.

RECUERDA

Una variable local no puede ser declarada como `static`.

2.1.3.1 Visibilidad y vida de las variables

Visibilidad, *scope* o **ámbito de una variable** son sinónimos. Visibilidad es la parte del código de una aplicación donde la variable es accesible y puede ser utilizada.

Al contrario que en otros lenguajes de programación, en Java las variables no pueden declararse fuera de una clase.

Por regla general, en Java, todas las variables que están dentro de un bloque (entre `{y}`) son visibles y existen dentro de dicho bloque. Las funciones miembro de una clase podrán acceder a todas las variables miembro de dicha clase, pero no a las variables locales de otra función miembro.

2.1.4 OPERADORES ARITMÉTICOS

Los operadores aritméticos son utilizados para realizar operaciones matemáticas.

Operador	Uso	Operación
+	A + B	Suma
-	A - B	Resta
*	A * B	Multiplicación
/	A / B	División
%	A % B	Módulo o resto de una división entera

Tabla 2.3. *Operadores aritméticos*

En el siguiente ejemplo se puede observar la utilización de operadores aritméticos:

```
int n1=2, n2;
n2=n1 * n1; // n2=4
n2=n2-n1; // n2=2
n2=n2+n1+15; // n2=19
n2=n2/n1; // n2=9
n2=n2%n1; // n2=1
```

2.1.5 OPERADORES RELACIONALES

Con los operadores relacionales se puede evaluar la igualdad y la magnitud. En la siguiente tabla A y B no son los operadores, sino los operandos; como se puede ver:

Operador	Uso	Operación
<	A < B	A menor que B
>	A > B	A mayor que B
<=	A <= B	A menor o igual que B
>=	A >= B	A mayor o igual que B
!=	A != B	A distinto que B
= =	A == B	A igual que B

Tabla 2.4. *Operadores relacionales*

En el siguiente ejemplo se puede observar la utilización de operadores relacionales:

```
int m=2, n=5;
boolean res;
res =m > n;//res=false
res =m < n;//res=true
res =m >= n;//res=false
res =m <= n;//res=true
res =m == n;//res=false
res =m != n;//res=true
```

2.1.6 OPERADORES LÓGICOS

Con los operadores lógicos se pueden realizar operaciones lógicas. En la siguiente tabla A y B no son los operadores, sino los operandos; como se puede ver:

Operador	Uso	Operación
&& o &	A&&B o A&B	A **AND** B. El resultado será `true` si ambos operandos son `true` y `false` en caso contrario.
|| o |	A || B o A | B	A **OR** B. El resultado será `false` si ambos operandos son `false` y `true` en caso contrario.
!	!A	**Not** A. Si el operando es `true` el resultado es `false` y si el operando es `false` el resultado es `true`.
^	A ^ B	A **XOR** B. El resultado será `true` si un operando es `true` y el otro `false`, y `false` en caso contrario.

Tabla 2.5. *Operadores lógicos*

En el siguiente ejemplo se puede observar la utilización de operadores lógicos:

```
int m=2, n=5;
boolean res;
res =m > n && m >= n;//res=false
res =!(m < n || m != n);//res=false
```

2.1.7 OPERADORES UNITARIOS O UNARIOS

Operador	Uso	Operación
~	~A	Complemento a 1 de A
-	-A	Cambio de signo del operando
--	A--	Decremento de A
++	A++	Incremento de A
!	!A	Not A (ya visto)

Tabla 2.6. *Operadores unitarios*

En el siguiente ejemplo se puede observar la utilización de operadores unitarios:

```
int m=2, n=5;
m++; // m=3
n--; // n=4
```

2.1.8 OPERADORES DE BITS

Operador	Uso	Operación
&	A & B	AND lógico. A AND B.
|	A | B	OR lógico. A OR B.
^	A ^ B	XOR lógico. A XOR B.
<<	A << B	Desplazamiento a la izquierda de A B bits rellenando con ceros por la derecha.
>>	A >> B	Desplazamiento a la derecha de A B bits rellenando con el BIT de signo por la izquierda.
>>>	A >>> B	Desplazamiento a la derecha de A B bits rellenando con ceros por la izquierda.

Tabla 2.7. *Operadores de bits*

En el siguiente ejemplo se puede observar la utilización de operadores de bits:

```
int num=5;
num = num << 1; // num = 10, equivale a num = num * 2
num = num >> 1; // num = 5, equivale a num = num / 2
```

2.1.9 OPERADORES DE ASIGNACIÓN

Operador	Uso	Operación
=	A = B	Asignación. Operador ya visto.
*=	A *= B	Multiplicación y asignación. La operación A*=B equivale a A=A*B.
/=	A /= B	División y asignación. La operación A/=B equivale a A=A/B.
%=	A %= B	Módulo y asignación. La operación A%=B equivale a A=A%B.
+=	A += B	Suma y asignación. La operación A+=B equivale a A=A+B.
-=	A -= B	Resta y asignación. La operación A-=B equivale a A=A-B.

Tabla 2.8. Operadores de asignación

En el siguiente ejemplo se puede observar la utilización de operadores de asignación:

```
int num=5;
num += 5; // num = 10, equivale a num = num + 5
```

2.1.10 PRECEDENCIA DE OPERADORES

CONSEJO

Utiliza paréntesis: de esa forma puedes dejar los programas más legibles y controlar las operaciones sin tener que depender de la precedencia.

La precedencia de operadores se resume en la siguiente tabla:

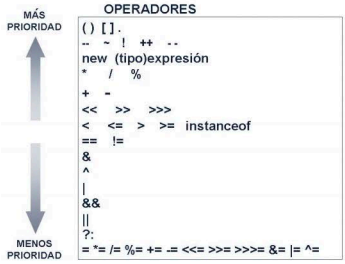


Figura 2.1. Precedencia de operadores

Imaginemos que se tiene un código como el siguiente:

```
int a = 4;
a = 5 * a + 3;
```

Se desea conocer el valor que tomará `a`. Para ello se mira en la tabla y se puede observar que el operador `*` tiene más precedencia que el operador `+`, con lo cual primero se ejecutará `5 * a`, y al resultado de esta operación se le sumará `3`. El resultado de la expresión será `23` y por lo tanto el valor de `a` será `23` al ejecutar este código.

2.1.11 EXPRESIONES

Una vez hemos visto las variables, las constantes, los operadores y la precedencia de operadores, diremos que una expresión es una combinación o asociación de los elementos anteriores de tal manera que en un lenguaje determinado se pueda evaluar.

Ejemplos de expresiones serían los siguientes:

- A+25-b/2
- n2+n1+15
- (a+b)/c

2.2 ESTRUCTURAS DE CONTROL. SECUENCIAL, CONDICIONAL Y DE REPETICIÓN

IMPORTANTE: LAS SENTENCIAS

Una expresión es una serie de variables/constantes/datos unidos por operadores (por ejemplo, `2*PI*radio`). Una sentencia es una expresión que acaba en `;` (por ejemplo, `area = 2*PI*radio;`).

2.2.1 ESTRUCTURA SECUENCIAL

La estructura secuencial se compone de un grupo de sentencias que se irán ejecutando una detrás de otra. El diagrama de flujo de una estructura secuencial sería el siguiente:

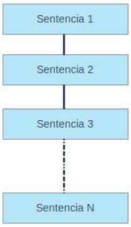


Figura 2.2. *Diagrama de flujo de una estructura secuencial*

Las estructuras secuenciales están formadas por instrucciones que no implican salto, como pueden ser:

- Una asignación.
- Una escritura o salida de datos.
- Una lectura o entrada de datos.
- Declaraciones de variables o constantes.

A continuación se muestra un código con varias sentencias en una estructura secuencial:

```
int n1=5;
int n2=30;
int suma=0;
suma=n1+n2;
System.out.println("LA SUMA ES: " + suma);
```

2.2.2 ESTRUCTURA CONDICIONAL

Entre las estructuras condicionales nos encontramos con la estructura `IF` y con la `SWITCH`.

2.2.2.1 Las estructuras if

En Java hay tres tipos de estructuras `if` (`if`, `if-else` e `if-elseif-else`), el formato de este tipo de estructuras es el siguiente:

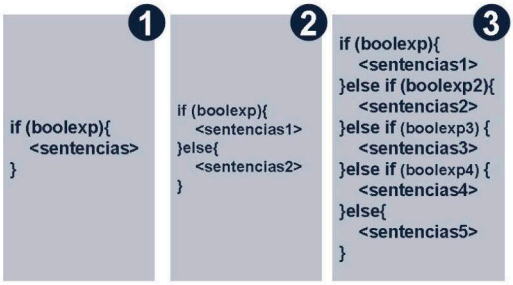


Figura 2.3. *Diferentes sentencias IF*

En los casos anteriores, `boolexp` es una expresión booleana que puede ser verdadera o falsa. Ejemplos de expresiones booleanas pueden ser (a > 20 o 2*PI*radio>30). Dependiendo de si es verdadera o falsa se ejecutarán o no unas sentencias. Como se puede apreciar, en el caso 3, sería producto de combinar o anidar un `if` dentro de otro.

En la siguiente figura se puede observar cómo se ejecutará el flujo del programa para los casos anteriores 1 y 2:

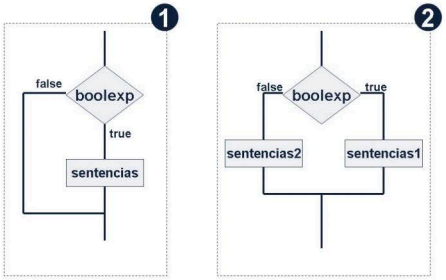


Figura 2.4. *Diagrama de flujo de las sentencias IF*

Un ejemplo de la utilización de estas estructuras es el siguiente:

```
int a = 4;
if (a = = 4) {
       System.out.println("La variable es igual a 4");
}
if (a > 5){
       System.out.println("La variable es mayor a 5");
}else{
       System.out.println("La variable es menor que 6");
}
if (a > 5){
       System.out.println("La variable es mayor a 5");
}else if(a = = 5){
       System.out.println("La variable es igual a 5");
}else{
       System.out.println("La variable es menor que 5");
}
```

Otro ejemplo anidando las sentencias `if`:

```
int matematicas = 4, lengua = 2;
if (matematicas >= 5){
       if (lengua >= 5){
              System.out.println("Enhorabuena");
       }else{
              System.out.println("No has aprobado todas las asignaturas");
       }
}else{
       System.out.println("No has aprobado todas las asignaturas");
}
```

2.2.2.2 Las estructuras switch

Cuando una expresión puede tener varios valores —y dependiendo del valor que tome hay que ejecutar una serie de sentencias—, hay dos posibilidades a la hora de programar: la primera es utilizar la estructura `switch`, la cual deja el código limpio y fácil de interpretar; otra opción es utilizar estructuras `if` para resolver este problema. El formato de esta estructura es el siguiente:

```
switch (expresión){
      case valor1: sentencias1; break;
      case valor2: sentencias2; break;
      case valor3: sentencias3; break;
      .......
      case valorn: sentenciasn; break;
      [default: sentenciasdef;]
}
```

Figura 2.5. *La sentencia SWITCH*

RECUERDA

Si no se escribe la sentencia `break`, el programa seguirá ejecutando las siguientes sentencias hasta encontrarse con un `break` o el fin del `switch`.

Un ejemplo de utilización de esta estructura es el que se muestra a continuación:

```
switch(posicion){
       case 1: System.out.println("ORO");break;
       case 2: System.out.println("PLATA");break;
       case 3: System.out.println("BRONCE");break;
       case 4: System.out.println("DIPLOMA");break;
       case 5: System.out.println("DIPLOMA");break;
       default: System.out.println("SIN PREMIO");break;
}
```

2.2.3 ESTRUCTURA ITERATIVA

Las estructuras iterativas, o bucles, son utilizadas cuando una o varias sentencias han de ser ejecutadas cero, una o más veces.

CUIDADO

Ten cuidado con los bucles infinitos. Los bucles infinitos son aquellos que no terminan nunca (aquellos cuya expresión booleana siempre es cierta, o `true`). En el caso de producirse un bucle infinito, el programa se seguirá ejecutando y se quedará "colgado" hasta que el usuario mate el proceso.

2.2.3.1 La estructura iterativa while

El bucle `while` se utiliza cuando se tiene que ejecutar un grupo de sentencias un número determinado de veces (0 o más veces). El formato de estructura del bucle `while` es el siguiente:

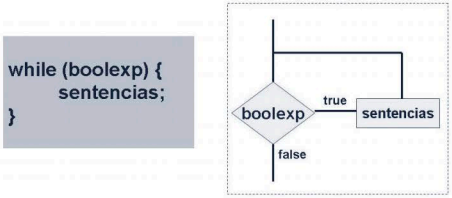


FIgura 2.6. *Estructura del bucle while*

Un ejemplo de utilización de esta estructura es el que se muestra a continuación:

```
int numero = 1;
while(numero<=10){ //bucle que cuenta hasta 10
       System.out.println(numero);
       numero++;
}
```

Este código anterior lo que hace es mostrar por pantalla los números del 1 al 10.

2.2.3.2 La sentencia iterativa do while

La estructura iterativa `do while` —o mejor llamada `until` en otros lenguajes— es una variante del `while`. En realidad cualquier estructura `do while` se puede transformar en una estructura `while`. El formato de estructura del bucle `do while` es el siguiente:

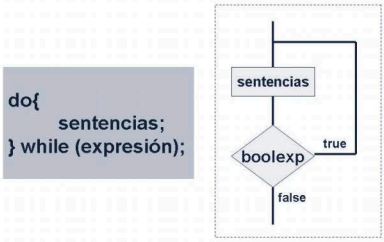


Figura 2.7. *Estructura del bucle do while*

Como se puede observar, esta estructura es igual a la anterior (`while`), lo único que ocurre es que la comprobación se hace al final del bucle, con lo cual siempre se ejecutarán las sentencias al menos una vez.

Un ejemplo de utilización de esta estructura es el que se muestra a continuación:

```
int numero = 1;
do{ //bucle que cuenta hasta 10
       System.out.println(numero);
       numero++;
}while(numero<=10);
```

Este ejemplo es igual al anterior, lo único que se ha cambiado es el `while` por el `do while`.

2.2.3.3 La estructura iterativa for

El bucle `for` se utiliza cuando se necesita ejecutar una serie de sentencias un número fijo y conocido de veces. La estructura tiene el siguiente formato:

```
For(inicialización;boolexp;incremento){
       sentencias;
};
```

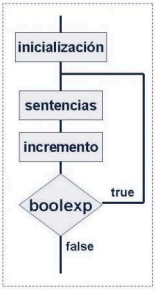


Figura 2.8. *Estructura del bucle for*

Fijándonos en la estructura podemos ver que la estructura `for` se puede conseguir utilizando el bucle `while`. En el primer ejercicio resuelto se puede ver cómo se realizaría esto.

Un ejemplo de utilización de esta estructura es el que se muestra a continuación:

```
int numero = 1;
for (numero=1;numero<=10;numero++){ //bucle que cuenta hasta 10
       System.out.println(numero);
}
```

TRUCO

Si queremos que en el ejemplo anterior el contador en vez de incrementarse se decremente, utilizaremos `numero--`. Si queremos que se incremente de 2 en 2, haremos entonces `numero+=2`.

2.2.4 OTROS TIPOS DE ESTRUCTURAS

CONSEJO

Se desaconseja el uso de las sentencias `break` salvo para la estructura `switch`. Las sentencias de salto dificultan la legibilidad de los programas y evitan que la programación sea estructurada.

2.2.4.1 La sentencia break

La sentencia **`break`**, ya vista anteriormente, sirve tanto para las estructuras de selección como para las estructuras de repetición. El programa, al encontrar dicha sentencia, se saldrá del bloque que está ejecutando.

2.2.4.2 La sentencia return

La sentencia **`return`** es otra forma de salir de una estructura de control. La diferencia con `break` y `continue` es que `return` sale de la función o método (procedimiento) que está ejecutando y permite devolver un valor. Por ejemplo:

```
return 5; //sale de la función o método y devuelve el valor 5.
```

2.2.4.3 Control de excepciones

El control de excepciones va a permitir al programador controlar la ejecución del programa evitando que este falle de forma inesperada.

La estructura del control de excepciones tiene el siguiente formato:

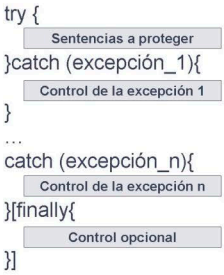


Figura 2.9. *Estructura del control de excepciones en Java*

El programa intentará proteger las sentencias situadas dentro del bloque **try**, en el caso de que ocurra un error se intentará controlar la excepción mediante los bloques **catch** (dependiendo de la excepción se ejecutará un bloque de código u otro). El bloque **finally** es opcional, pero, en caso de existir, este se ejecutará siempre.

Se va a ver una muestra del control de excepciones en los dos siguientes ejemplos.

Veamos el siguiente programa:

```
class Test
{
    public static void main(String [] args)
    {
    int a=10, b=0, c;
    c=a/b;
    System.out.println("Resultado:"+c);
    }
}
```

Como todo el mundo sabe, el dividir por cero es una indeterminación, con lo cual este tipo de operaciones provocará en el programa el siguiente error, lo que causará una finalización anormal del mismo:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at Test.main(Test.java:6)
Presione una tecla para continuar . . .
```

Una forma de controlar esta situación será la siguiente:

```
class Test
{
    public static void main(String [] args)
    {
    int a=10, b=0, c;

    try{
           c=a/b;
    }
    catch(ArithmeticException e){
           System.out.println("Error: "+e.getMessage());
           return;
    }
    System.out.println("Resultado:"+c);
    }
}
```

De esta manera se regula la ejecución del programa y este finalizará de una forma controlada.

2.3 FUNCIONES Y PROCEDIMIENTOS. MÉTODOS DE LAS CLASES

En esta sección se va a trabajar en profundidad con el paso de parámetros por valor y por referencia, los miembros `static`, así como con los métodos de instancia y de clase. Es importante que el alumno comprenda y sepa diferenciar estos miembros y ambos métodos.

2.3.1 MIEMBROS ESTÁTICOS (STATIC) DE UNA CLASE/MIEMBROS DE CLASE

En Java no existen variables globales, por lo tanto, si queremos utilizar una variable única y que puedan utilizar todos los objetos de una clase deberemos declararla como estática (`static`).

RECUERDA

A diferencia de los miembros normales o miembros de instancia, los miembros de clase tienen la cláusula `static` y todos los objetos de la misma clase compartirán dichos miembros.

Veamos cómo funcionan los atributos estáticos de una clase con el siguiente ejemplo:

```
public class cohete{
    private static int numcohetes=0;
    cohete(){ numcohetes++; }
    public int getcohetes(){ return numcohetes;}
}
```

Tenemos una clase, la cual tiene un miembro estático. Esta variable `numcohetes` almacenará el número de objetos `cohete` que se van creando.

```
public class testestaticos {
    public static void main(String[] args) {
          cohete c1 = new cohete();
          cohete c2 = new cohete();
          cohete c3 = new cohete();
          System.out.println(c1.getcohetes());
          System.out.println(c3.getcohetes());
 }
}
```

Cuando desde otra clase, por ejemplo la anterior, se crean varios objetos de la clase `cohete` (3 objetos) y se llama al método `getcohetes()`, ¿qué valores devolverá dicho método?

```
          System.out.println(c1.getcohetes());
          System.out.println(c3.getcohetes());
```

La solución es 3 en ambas llamadas. La variable `numcohetes` se inicializa a 0 solo una vez (cuando se crea el objeto `c1`). Cuando se crean los objetos `c2` y `c3` no se vuelve a inicializar, pues ya existe y es estática, solo se incrementa.

Al haber definido `numcohetes` como `private`, no es posible desde nuestra clase `testestaticos` acceder a `c1.numcohetes`.

RECUERDA

Los miembros o atributos de instancia son aquellos que no son `static`.

2.3.2 MÉTODOS DE INSTANCIA Y DE CLASE

Los métodos de una clase son una abstracción del comportamiento de la misma. Los **algoritmos** formarán parte de los métodos y contendrán la lógica de la aplicación que queramos desarrollar.

Podemos dividir los métodos en dos bloques:

- **Métodos de instancia**. Son aquellos utilizados por la instancia.
- **Métodos de clase**. Son aquellos comunes para una clase. Un método por clase.

RECUERDA

Miembros o atributos de instancia y de clase son análogos a métodos de instancia y de clase.

2.3.3 MÉTODOS DE INSTANCIA

Los métodos de instancia son, por así decirlo, los llamados "métodos comunes". Cada instancia u objeto tendrá sus propios métodos independientes del mismo método de otro objeto de la misma clase.

```
public class cuadrado{
    private int lado;
    cuadrado(int l){ this.lado = l; }
    public int getArea(){ return lado*lado; }
}
```

En el anterior ejemplo podemos ver un ejemplo de un método de instancia.

RECUERDA

Los métodos de instancia pueden acceder a los miembros de instancia y también a los miembros de clase.

La siguiente clase, atendiendo a la regla anterior, compilará sin problemas:

```
class test {
    public static int var;
    public int var2;
    public void prueba(){
           var = 3;
           var2 = 5;
    }
}
```

No obstante, en vez de utilizar la línea:

```
           var = 3;
```

Quizás hubiese sido más correcto utilizar la siguiente:

```
           test.var = 3;
```

La llamada a un método de instancia sería la siguiente:

```
test t = new test();
t.prueba();
```

2.3.4 MÉTODOS ESTÁTICOS O DE CLASE

Recuerda las siguientes reglas.

- 1. Los métodos `static` no tienen referencia `this`.
- 2. Un método `static` no puede acceder a miembros que no sean `static`.
- 3. Un método no `static` puede acceder a miembros `static` y no `static`.

Veamos alguna de estas reglas en un pequeño programa:

```
public class Test {
    public int dato=0;
    public static int datostatico=0;
    public void metodo(){this.datostatico++;}
    public static void metodostatico(){
           this.datostatico++; // Esto da error al compilar
           datostatico++;
    }
    public static void main(String[] args) {
           dato++; // Esto da error al compilar
           datostatico++;
```

```
        metodostatico();
        metodo(); // Esto da error al compilar
    }
}
```

Veremos las razones por las cuales las líneas resaltadas en negrita dan error de compilación.

```
this.datostatico++; // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 1 (los métodos `static` no tienen referencia `this`).

```
dato++; // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 2 (un método `static` no puede acceder a miembros que no sean `static`) dado que `dato` no es `static`.

```
metodo(); // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 2 (un método `static` no puede acceder a miembros que no sean `static`), dado que el método `metodo()` no es `static`. Sin embargo, según la regla 3, el método `metodo()` puede acceder al dato `datostatico`, dado que este método no es estático.

RECUERDA

Los métodos de clase o `static` NUNCA pueden acceder a los miembros de instancia.

Método	Llamada	Declaración	Acceso
Clase	`Clase.metodo(parámetros)`	`static`	Miembros de clase.
Instancia	`Instancia.metodo(parámetros)`		Miembros de clase y de instancia.

Tabla 2.9. *Tabla resumen*

Un ejemplo de los métodos de clase son las funciones de la librería `java.lang.Math`, las cuales pueden ser llamadas anteponiendo el nombre de la clase `Math`. Un ejemplo de llamada a una de estas funciones es, por ejemplo:

```
Math.cos(angulo);
```

Como se puede observar, se antepone el nombre de la clase (`Math`) al del método `cos`.

Algunos métodos estáticos (los más utilizados) de la clase `Math` son los siguientes:

Método	Descripción

`static int abs(int a)`
`static long abs(long a)`
`static double abs(double a)`
`static float abs(float a)`
	Devuelve el valor absoluto del parámetro pasado.

`static int max(int a, int b)`
`static long max(long a, long b)`
`static double max(double a, double b)`
`static float max(float a, float b)`
	Devuelve el mayor de los valores `a` o `b`.

`static int min(int a, int b)`
`static long min(long a, long b)`
`static double min(double a, double b)`
`static float min(float a, float b)`
	Devuelve el menor de los valores `a` o `b`.

`static double pow(double a, double b)`	Potencia de un número. Devuelve el valor de `a` elevado a `b`.
`static double random()`	Números aleatorios. Devuelve un número aleatorio de tipo `double` entre cero y uno (este último no incluido).
`static int round(float a)` `static long round(double a)`	Redondeo. Redondea el parámetro `a` al valor entero más cercano.

***Tabla 2.10.** Métodos de la clase Math*

Además de los métodos vistos, la clase `Math` tiene un sinfín de funciones trigonométricas, además de muchas otras funciones.

2.3.5 PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

Es común pasar parámetros a los métodos, salvo que sean métodos para inicializar o finalizar el objeto. Existen ocasiones en las que necesitamos que estas variables que pasamos como parámetros cambien su valor una vez ejecutado el método si este las ha modificado. Si es una variable, se puede solucionar con la sentencia `return`, pero imagínate que queremos pasar 5 variables a un método y conservar los valores si este los modifica. En ese caso, con `return` solamente podríamos obtener una variable modificada. Por lo tanto, deberemos utilizar el paso de parámetros por referencia.

En resumen:

- **Paso de parámetros por valor**. Los parámetros se copian en las variables del método. Las variables pasadas como parámetro no se modifican.
- **Paso de parámetros por referencia**. Las variables pasadas como parámetro se modifican puesto que el método trabaja con las direcciones de memoria de los parámetros.

Un ejemplo de esto que se acaba de explicar es el siguiente:

```
public class testparam {
    public static void cambiar(int x){
          x++;
    }
    public static void cambiar2(int[] par){
          par[0]++;
    }
    public static void main(String[] args) {
          int x = 3;
          int []arrx={3};
          cambiar(x);
          System.out.println(x);
          cambiar2(arrx);
          System.out.println(arrx[0]);
 }
}
```

Este programa dará como salida los valores 3 y 4. En la función `cambiar2` se pasa un *array* en vez de una variable. La diferencia entre un *array* de enteros y una variable entera es que un *array* de enteros es una dirección de memoria donde de manera consecutiva se almacenarán una serie de valores enteros. Los *arrays* o vectores se estudiarán en profundidad más adelante, en el capítulo 3.

Representado de forma gráfica, el comportamiento del programa es el siguiente:

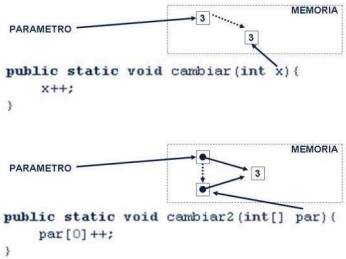


Figura 2.10. *Parámetros por valor y por referencia*

Como puedes ver, en la función `cambiar`, lo que se hace es que se copia el contenido del parámetro a la variable `x` del método. Sin embargo, en la función `cambiar2`, aunque se hace lo mismo, lo que cambia es que el valor que contiene el parámetro es a su vez una dirección de memoria. Con lo cual, cada cambio en la variable `par` del método repercutirá en un cambio del parámetro pasado por referencia.

2.4 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:
- **a)** El valor `null` no se considera como un literal.
- **b)** En Java las variables no pueden declararse fuera de una clase.
- **c)** Con un constructor de copia se inicializa un objeto asignándole los valores de otro objeto diferente de la misma clase.
- **d)** El método `gc()` de la clase `System` hace una llamada al garbage collector.

2 ¿Cuál de las siguientes afirmaciones es falsa?:
- **a)** El operador == tiene mayor prioridad que el operador >.
- **b)** Una expresión es una serie de variables/constantes/datos unidos por operadores.
- **c)** Los programas o aplicaciones en Java se componen de una serie de ficheros `.class` que son ficheros en *byte-code*.
- **d)** `~A` es el complemento `a 1` de `A`.

3 ¿Cuál de las siguientes afirmaciones es falsa?:
- **a)** Las variables miembros de una clase y las variables locales se inicializan por defecto.
- **b)** Se desaconseja el uso de las sentencias `break` salvo para la estructura `switch`.
- **c)** La aplicación se ejecuta desde el método o procedimiento principal `main()` situado en una clase o fichero principal.
- **d)** Una variable local no puede ser declarada como `static`.

4 ¿Cuál de las siguientes afirmaciones es falsa?:
- **a)** `A XOR B`. El resultado será `true` si un operando es `true` y el otro `false`, y `false` en caso contrario.
- **b)** Las constantes se declaran utilizando la palabra `static`.
- **c)** La sentencia `break` solamente sirve tanto para las estructuras de selección.
- **d)** La estructura iterativa `do while` se denomina `until` en otros lenguajes de programación.

5 ¿Cuál de las siguientes afirmaciones es falsa?:
- **a)** Un literal puede ser el valor `null`.
- **b)** En un bucle `while`, las sentencias de dentro del mismo se ejecutarán al menos una vez.
- **c)** En el paso de parámetros por valor, los parámetros se copian en las variables del método.
- **d)** En Java no existen variables globales; por lo tanto, si queremos utilizar una variable única y que puedan utilizar todos los objetos de una clase, deberemos declararla como estática (`static`).

3 Estructura de la información

En este capítulo se estudiará en profundidad todo lo relativo a información y datos. Veremos estructuras estáticas y dinámicas, además de cómo Java gestiona la memoria.

3.1 ESTRUCTURAS ESTÁTICAS

Las **estructuras estáticas**, al contrario que las dinámicas, no utilizan punteros. Las más utilizadas son los *arrays*, las matrices y las cadenas de caracteres. Estas tres estructuras son muy útiles y nos van a servir para resolver una gran cantidad de problemas.

Las estructuras estáticas se utilizan cuando se sabe a ciencia cierta el número de elementos que van a contener. Por ejemplo, si queremos hacer un *array* con los compañeros de clase, un *array* con la clasificación de la liga de fútbol, una tabla con los resultados de los emparejamientos de un torneo de tenis, etc.

Cuando no sabemos el número de elementos que va a tener la estructura, muchas veces es mejor resolver el problema utilizando una estructura dinámica que puede crecer y menguar según la necesidad.

3.1.1 ARRAYS O VECTORES

Hasta ahora, en ninguno de los ejercicios o ejemplos que se han ido viendo durante los temas anteriores ha habido una especial necesidad de almacenar muchos valores: se utilizaban variables para guardar el color, la edad, la cantidad, etc. Imaginemos que nos piden realizar un programa que almacene la temperatura de 100 ciudades españolas y luego saque la temperatura media nacional. No parece operativo tener 100 variables en nuestro programa: con solo escribir los nombres en el código, el número de líneas se dispara. ¿Y si nos dicen que se va a aumentar el número de ciudades a 200? La solución a esto son los *arrays* o vectores (son sinónimos).

RECUERDA

En Java se pueden crear vectores o *arrays* de tipos básicos (`boolean`, `int`, `byte`, etc.) y también *arrays* de objetos. De esa manera se pueden almacenar varios valores en cada posición de memoria.

Un *array* se compone de una serie de posiciones consecutivas en memoria.

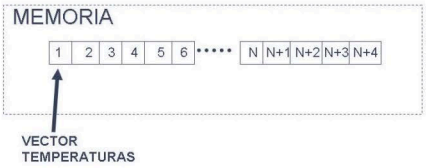


Figura 3.1. *Almacenamiento del vector temperaturas en memoria*

A los vectores se accede mediante un subíndice, si por ejemplo nuestro vector anterior se llama `temperaturas` y se quiere acceder a la posición `N`, habrá que escribir `temperaturas[N]` en el programa para obtener la información de esa posición de memoria. `N` puede ser una variable o bien un valor concreto.

3.1.1.1 Declaración de vectores

En Java se pueden **declarar vectores** de dos formas diferentes. Podemos declarar nuestro vector de temperaturas de las siguientes formas:

```
byte[] temperaturas;
byte temperaturas[];
```

Como puede observarse, en ningún momento se ha dado el tamaño de la matriz, lo único que se ha especificado es el tipo de los elementos que va a albergar dicha matriz.

3.1.1.2 Creación de vectores

Java trata los vectores como si fuesen objetos, por lo tanto la creación de nuestro vector `temperaturas` será del siguiente modo:

```
temperaturas = new byte[100];
```

Lo que implica reservar en memoria 100 posiciones de tipo `byte`.

RECUERDA

En los vectores, cuando se reservan `N` posiciones de memoria, los datos se almacenarán en las posiciones `0, 1, ····· N-1`.

El tamaño también se le puede asignar mediante una variable de la siguiente forma:

```
int v=100;
byte[] temperaturas;
temperaturas = new byte[v];
```

También es muy común ver en los programas este tipo de declaraciones:

```
int v=100;
byte[] temperaturas = new byte[v];
```

Como se puede ver, se funden la segunda y tercera líneas de código del ejemplo anterior en una sola.

3.1.1.3 Inicialización de vectores

Si no se especifica ningún valor, los elementos de un vector se inicializan automáticamente a unos valores predeterminados (variables numéricas a 0, objetos a `null`, booleanas a `false` y caracteres a `'\u0000'`).

También es posible inicializarlo con los valores que desee el programador:

```
byte[] temperaturas={10,11,12,11,10,9,18,19,14,13,15,15};
```

En este ejemplo anterior se ha creado un vector de 12 posiciones del tipo `byte` con los valores especificados.

3.1.1.4 Métodos de los vectores

Como se ha dicho anteriormente, Java maneja los vectores como si fueran objetos, por lo tanto existen métodos heredados de la clase `Object`, que está en el paquete `java.lang`:

- **`equals`**. Permite discernir si dos referencias son el mismo objeto.
- **`clone`**. Duplica un objeto.

Un ejemplo de utilización de estos métodos es el siguiente:

```
byte[] temperaturas1={10,11,12,11,10,9,18,19,14,13,15,15};
byte[] temperaturas2=(byte[])temperaturas1.clone();
byte[] temperaturas3=temperaturas1;

if (temperaturas1.equals(temperaturas2)){
            System.out.println("temperaturas1==temperaturas2");
}else{
            System.out.println("temperaturas1!=temperaturas2");
}
if (temperaturas1.equals(temperaturas3)){
            System.out.println("temperaturas1==temperaturas3");
}else{
            System.out.println("temperaturas1!=temperaturas3");
}
```

En el ejemplo anterior, el programa mostrará los siguientes literales: "`temperaturas1!=temperaturas2`" y "`temperaturas1==temperaturas3`" porque en el primer caso, aunque los datos son los mismos, el objeto es diferente; y en el segundo caso, al asignar `temperaturas3=temperaturas1` hace que `temperaturas3` referencie al mismo objeto (apunta al mismo lugar en la memoria, no se duplican los datos), y en ese caso el método `equals` sí da como resultado `true`.

3.1.1.5 Utilización de los vectores

Un ejemplo de utilización de los vectores es el siguiente programa:

```
public class temperaturas {
    private static int[] temperaturas1;
    final static int POS=10; //número de posiciones del array
    public static void main(String[] args) {
    int dato=0;
    int media=0;
    temperaturas1 = new int[POS];
    for (int i=0;i<POS;i++){ //leer los valores de temperatura
          try{
                System.out.println("Introduzca Temperatura:");
                String sdato = System.console().readLine();
                dato = Integer.parseInt(sdato);
          }catch(Exception e){
                System.out.println("Error en la introducción de datos");
          }
          temperaturas1[i]=dato;
    }
    for (int i=0;i<POS;i++){//hacer la media
```

```
            media = media + temperaturas1[i];
        }
        media = media / POS;
        System.out.println("La media de temperaturas es "+media);
        }
}
```

En el programa anterior se leen las temperaturas de una serie de ciudades por teclado y luego se muestra la media de temperaturas. Como se puede observar, se crea la constante `POS`, la cual contiene el número de temperaturas a registrar. En el ejemplo está definida con valor 10, pero se puede aumentar o disminuir su valor y el programa funcionará sin modificar más el código.

3.1.2 ARRAYS MULTIDIMENSIONALES O MATRICES

Tratar con matrices en Java es parecido a tratar con vectores. Por ejemplo, una matriz de enteros de dos dimensiones con 5 filas y 8 columnas se crearía de la siguiente manera:

```
int[][] matriz = new int[5][8];
```

La inicialización del *array* en el momento de la declaración se hará del siguiente modo:

```
int [][] matriz = {{1,4,5},{6,2,5}};
```

En el ejemplo anterior se ha creado un *array* de 2 filas y 3 columnas.

```
System.out.println(matriz.length);
System.out.println(matriz[0].length);
```

El código anterior muestra en la primera línea el número de filas de la matriz creada anteriormente (2), y en la segunda línea, el número de columnas de la fila 0 de la matriz (3).

	COLUMNAS								
FILAS	M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]	M[0][6]	M[0][7]
	M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]	M[1][6]	M[1][7]
	M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]	M[2][6]	M[2][7]
	M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]	M[3][6]	M[3][7]
	M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]	M[4][6]	M[4][7]

Figura 3.2. *Matriz de datos*

El acceso a la matriz se haría igual que cuando se ha trabajado con vectores (`matriz[fila][columna]`). Imaginemos que queremos almacenar en cada celda de la matriz la suma de la posición de la columna y la fila. El resultado sería el siguiente:

```
for (int i=0;i<5;i++){
      for (int j=0;j<8;j++){
            matriz[i][j]=i+j;
      }
}
```

3.1.3 LAS CADENAS DE CARACTERES

RECUERDA

Las cadenas de caracteres en Java se tratan como objetos de la clase `String`.

Una cadena de caracteres es un vector, o *array*, de elementos de tipo `char`.

```
char[] nombre1={'p','e','p','e'};
char[] nombre2={112,101,112,101};
char[] nombre3=new char[4];
```

En el ejemplo anterior, las variables `nombre1` y `nombre2` contienen exactamente lo mismo dado que internamente Java almacena los caracteres con sus símbolos ASCII correspondientes (a la `'p'` le corresponde el 112 y a la `'e'` el 101). La variable `nombre3` se ha creado como una cadena de 4 caracteres pero todavía no se ha inicializado y, por tanto, sus 4 posiciones contendrán el valor `'\0'`.

RECUERDA

Para comprobar que las cadenas de caracteres en Java se tratan como objetos de la clase `String` prueba a hacer lo siguiente:

```
System.out.println("HOLA".length());
```

La anterior línea muestra la longitud del *string* (cadena de caracteres), que en este caso sería 4.

3.1.3.1 La clase String

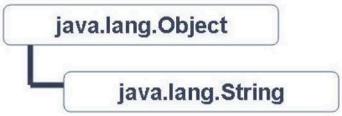


Figura 3.3. *La clase string desciende de Object*

La clase `String` pertenece al paquete `java.lang` y proporciona todo tipo de operaciones con cadenas de caracteres. Esta clase ofrece métodos de conversión a cadena de números, conversión a mayúsculas, minúsculas, reemplazamiento, concatenación, comparación, etc.

RECUERDA

Aparte de todos los siguientes métodos, el propio lenguaje Java ofrece el operador de concatenación +. Un ejemplo de utilización es:

```
System.out.println("Longitud de la cadena HOLA: "+"HOLA".length());
```

`String(String dato)`

Constructor de la clase `String`:

```
String cad1 = "Pepe";
String cad2 = new String("Emma");
String cad3 = new String(cad2);
```

Las anteriores tres líneas de código crean objetos de la clase `String`. Nótese como el objeto `cad3` está creado a partir del objeto `cad2` y contendrá los mismos datos: `"Emma"`.

`int length()`

Muestra la longitud de un objeto de la clase `String`:

```
String cad1 = "CHELO";
System.out.println(cad1.length());
```

El código anterior muestra la longitud del objeto `string cad1 (5).`

`String concat(String s)`

Devuelve un objeto fruto de la concatenación/unión de un objeto `String` con otro.

```
String cad1 = "Andy";
cad1=cad1.concat(" Rosique");
System.out.println(cad1);
```

El código anterior concatena las cadenas `"Andy"` y `"Rosique"`, y muestra por pantalla el resultado de la concatenación (`"Andy Rosique"`).

`String toString()`

Devuelve el propio `String`:

```
String cad1 = "Emilio";
String cad2 = " Anaya";
System.out.println(cad1.toString()+cad2.toString());
```

El código anterior aprovecha el operador de concatenación `"+"` para mostrar por pantalla la cadena `"Emilio Anaya"`.

`int compareTo(String s)`

Compara el objeto `String` con el objeto `String` pasado como parámetro y devuelve un número:

- < 0 Si es menor el *string* desde el que se hace la llamada al `String` pasado como parámetro.
- = 0 Si es igual el *string* desde el que se hace la llamada al `String` pasado como parámetro.
- > 0 Si es mayor el *string* desde el que se hace la llamada al `String` pasado como parámetro.

El método va comparando letra a letra ambos `String` y si encuentra que una letra u otra es mayor o menor que otra, deja de comparar.

```
String cad1 = "EMMA";
String cad2 = "MARIA";
System.out.println(cad1.compareTo("emma"));
System.out.println(cad1.compareTo("EMMA"));
System.out.println(cad1.compareTo("EMMA MORENO"));
System.out.println(cad2.compareTo("MARIA AMPARO"));
System.out.println(cad2.compareTo("MAREA"));
```

El anterior código mostrará por pantalla los siguientes datos: `-32, 0, -7, -7 y 4`.

¡CUIDADO!

El método `compareTo` distingue mayúsculas de minúsculas. Las mayúsculas están antes por orden alfabético que las minúsculas, por lo tanto `'A'` es menor que `'a'`.

```
boolean equals()
```

Este método sirve para comparar el contenido de dos objetos del tipo `String`.

```
String cad1="EMMA";
String cad2=new String("EMMA");
if (cad1.equals(cad2)){
    System.out.println("SON IGUALES");
}else{
    System.out.println("SON DIFERENTES");
};
```

El código anterior mostrará por pantalla `"SON IGUALES"`.

```
String trim()
```

Elimina los espacios en blanco que contenga el objeto `String` al principio y final del mismo.

```
String cad1 = " MAYKA ",cad2 = cad1.trim();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena `"MAYKA"`.

```
String toLowerCase()
```

Convierte las letras mayúsculas del objeto `String` en minúsculas.

```
String cad1 = "PEDRO ruiz",cad2 = cad1.toLowerCase();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena `"pedro ruiz"`.

```
String toUpperCase()
```

Convierte las letras minúsculas del objeto `String` en mayúsculas.

```
String cad1 = "MATI álvarez",cad2 = cad1.toUpperCase();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena `"MATI ÁLVAREZ"`.

```
String replace(char car, char newcar)
```

Reemplaza cada ocurrencia del carácter `car` por el carácter `newcar`.

```
String cad1 = "JUAN SUAREZ",cad2 = cad1.replace('U','O');
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena `"JOAN SOAREZ"`.

```
String substring(int i, int f)
```

Este método devuelve un nuevo objeto `String` que será la subcadena que comienza en el carácter `i` y termina en el carácter `f` (el carácter `f` no se muestra). Si no se especifica el segundo parámetro, devolverá hasta el final de la cadena.

```
String cad1 = "JUAN CARLOS MORENO";
System.out.println(cad1.substring(5,11));
System.out.println(cad1.substring(12));
```

El código anterior mostrará por pantalla las cadenas `"CARLOS"` y `"MORENO"`.

```
boolean startsWith(String cad)
```

Este método devuelve `true` si el objeto `String` comienza con la cadena `cad`, en caso contrario devuelve `false`.

```
String cad1 = "MAYKA MORENO";
System.out.println(cad1.startsWith("JUAN"));
System.out.println(cad1.startsWith("MAY"));
```

El código anterior mostrará por pantalla `false` y `true`.

```
boolean endsWith(String cad)
```

Este método devuelve `true` si el objeto `String` termina con la cadena `cad`, en caso contrario devuelve `false`.

```
String cad1 = "MARIA AMPARO";
System.out.println(cad1.endsWith("paro"));
System.out.println(cad1.endsWith("PARO"));
System.out.println(cad1.endsWith("ARIA"));
```

El código anterior mostrará por pantalla `false`, `true` y `false`. La primera vez muestra `false` porque, aunque `'p'` y `'P'` son la misma letra, Java las trata de manera diferente al ser dos símbolos ASCII distintos.

`char charAt(int pos)`

Devuelve el carácter del objeto `String` que se especifica en el parámetro `pos`.

```
String cad1 = "AMPARO HEREDIA";
System.out.println(cad1.charAt(0)+" "+cad1.charAt(7));
```

El código anterior mostrará por pantalla la cadena `"A H"`.

¡CUIDADO!

Si en la función `charAt` se utiliza un índice que no está entre los valores `0` y `length()-1`, Java lanzará una excepción.

`int indexOf(int c) y int indexOf(String s)`

Este método admite dos tipos de parámetros y nos permite encontrar la primera ocurrencia de un carácter o una subcadena dentro de un objeto del tipo `String`. En el caso de que no sea encontrado el carácter o la subcadena, este método devolverá el valor `-1`.

```
String cad1 = "EMMA MORENO";
System.out.println(cad1.indexOf('M'));
System.out.println(cad1.indexOf('J'));
System.out.println(cad1.indexOf("MO"));
System.out.println(cad1.indexOf("MI"));
```

El código anterior mostrara por pantalla el siguiente resultado: `1`, `-1`, `5` y `-1`.

`char[] toCharArray()`

Este método devuelve un vector o *array* de caracteres a partir del propio objeto `String`.

```
String cad1 = "LORO FELIPE";
char cad2[]=cad1.toCharArray();
```

El código anterior creará un *array* de caracteres `cad2` que contendrá la cadena `"LORO FELIPE"` contenida en el objeto `String cad1`.

`String valueOf(int dato)`

Convierte un número a un objeto `String`. La clase `String` es capaz de convertir los tipos primitivos `int`, `long`, `float` y `double`.

```
int edad1=6;
String str=String.valueOf(edad1);
float edad2=6;
str=String.valueOf(edad2);
```

```
long edad3=6;
str=String.valueOf(edad3);
double edad4=6.5;
str=String.valueOf(edad4);
```

¿CÓMO CONVERTIR UN STRING EN UN NÚMERO?

```
String snumero=" 6 ";
int numero=Integer.parseInt(snumero.trim());
//int numero=Integer.parseInt(snumero);
```

Con el código anterior es posible convertir un objeto `String` en un número entero. La tercera línea de código está comentada porque lanzaría una excepción dado que no se han limpiado los espacios a derecha e izquierda de la cadena y contiene caracteres no numéricos.
Si el número es un decimal y no se quieren perder los decimales, se utilizaría el siguiente código:

```
String snumero=" 6.5 ";
double numero=Double.valueOf(snumero).doubleValue();
```

3.2 ESTRUCTURAS DINÁMICAS Y DATOS ESTRUCTURADOS

Las estructuras dinámicas en Java generalmente se implementan mediante un conjunto de **nodos**, los cuales están divididos en dos partes: una donde se almacenan datos y otra donde existe un puntero que apunta, o bien a otro nodo de la lista, o bien a `null` o nulo (no apunta a ninguna parte).

En nuestro programa, la estructura básica `nodo` se representa mediante una clase:

```
class Nodo {
int dato;
Nodo sig;
}
```

Obviamente, en Java, cuando queremos agrupar una serie de datos relacionados entre sí, utilizamos una clase (en otros lenguajes de programación se utilizarán `struct`, registros, etc., que son estructuras parecidas).

Visualmente, un nodo sería una cosa parecida a esta:

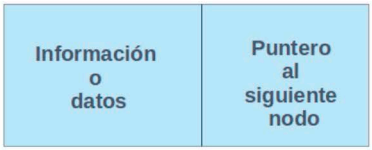


Figura 3.4. *Estructura de un nodo*

En nuestro caso anterior, para simplificar hemos hecho que los datos solamente sean un número entero, pero podrían tener múltiples datos incluyendo hasta otras clases si fuese necesario.

Como hemos dicho, el puntero al siguiente nodo puede apuntar a otro nodo o bien a `null`. Gráficamente, una estructura dinámica de datos podría ser algo parecido a lo siguiente:

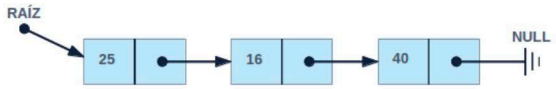


Figura 3.5. *Estructura dinámica de datos*

En la figura anterior podemos distinguir varios nodos conectados unos con otros, y el último de ellos apuntaría a `null` (a nada). También podemos distinguir un puntero o nodo inicial que se llama `raíz`. La raíz será el primer nodo de toda estructura y apuntará al primero de dicha estructura. En el caso de que inicialicemos la estructura, obviamente, apuntará a `null` porque no hay datos.

En Java, dicha inicialización se realiza de la siguiente forma:

```
private Nodo raiz; // se crea el primer nodo raíz

public Estructura () {
// Este es el método de inicialización de la estructura. Se ejecuta
// al comienzo del programa y hace que raíz apunte a null
raiz=null;
}
```

Gráficamente, dicha inicialización quedaría de la siguiente forma:

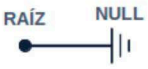


Figura 3.6. *Inicialización de la raíz*

Para crear la estructura anterior, el proceso paso a paso sería el siguiente:

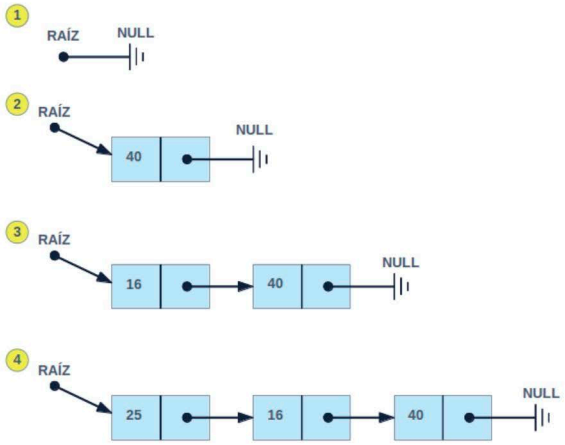


Figura 3.7. *Proceso de creación de una estructura dinámica*

Como se puede ver en la figura anterior, el primer paso es la inicialización y en los pasos siguientes se va insertando un nodo detrás de otro. En esta estructura, como en muchas otras, se puede ver que se insertan los elementos por la raíz.

Si los elementos se insertan por un lado y se extraen por el mismo lado, la estructura se llama "pila". El último en entrar es el primero en salir (*Last In First Out* - LIFO).

En el caso de que se inserte por la raíz y se extraigan por el final, se le llamará "cola". El primero en entrar es el primero en salir (*First In First Out* - FIFO).

RECUERDA

Las pilas y las colas son las estructuras dinámicas más sencillas, pero ten en cuenta que existen muchas más: listas ordenadas, listas doblemente ordenadas, árboles, etc.

A continuación se estudiarán en profundidad algunas de estas estructuras:

3.2.1 PILAS

Como hemos dicho, las pilas son estructuras donde se almacenan datos por un lado y se extraen los datos por el mismo lado, de tal manera que el último en entrar será el último en salir. A las dos funciones para insertar y para extraer las llamaremos **push** (insertar) y **pop** (extraer).

Veamos en profundidad el código de la función **push**:

```
public void push(int d) {
   Nodo nuevo;
nuevo = new Nodo();
nuevo.dato = d;
if (raiz==null){
nuevo.sig = null;
raiz = nuevo;
}else{
nuevo.sig = raiz;
raiz = nuevo;
}
}
```

Como se puede ver en el código, existen dos casos:

- Caso 1. En el primer caso la pila está vacía y por lo tanto apunta a `null` (**`raiz==null`**).
- Caso 2. En el segundo caso, la pila **al menos tiene un elemento**.

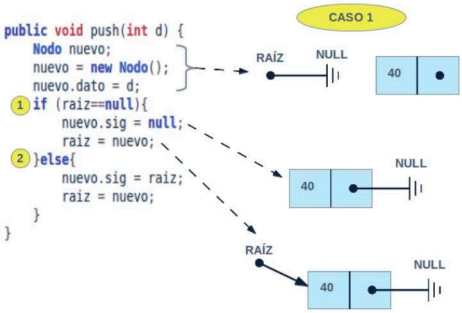


Figura 3.8. *Caso 1 de la función push*

En este primer caso, la raíz apunta a `null` y por lo tanto lo que tenemos que hacer es que el nuevo nodo apunte a `null`, dado que no hay ningún nodo al que apuntar. Una vez realizado esto, la raíz apuntará al nuevo nodo creado.

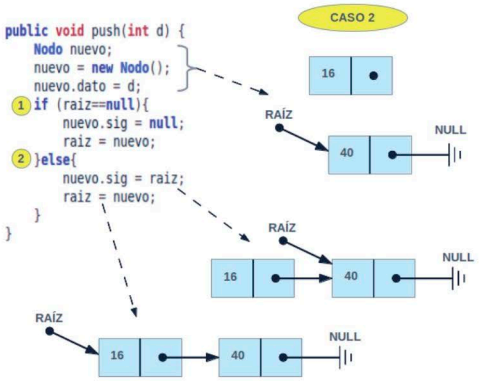


Figura 3.9. *Caso 2 de la función push*

En este caso, el nuevo nodo creado tendrá que apuntar al primero de la lista (donde apunta raíz) y una vez realizado esto la raíz apuntará al nuevo nodo creado. De esa manera lo que hemos hecho es intercalar el nuevo nodo entre la raíz y el nodo existente.

La otra función importante de este tipo de estructuras es la función `pop`. Esta función, o bien extrae dato a dato por la raíz, o bien devuelve un valor: en nuestro caso es `Integer.MAX_VALUE`, el mayor valor soportable en un dato de tipo `Integer` (este hecho indicará que ya no existen más datos que se puedan extraer).

El código de la función `pop` será el siguiente:

```
public int pop ()
{
if (raiz!=null){
int informacion = raiz.dato;
raiz = raiz.sig;
return informacion;
}else{
return Integer.MAX_VALUE;
}
```

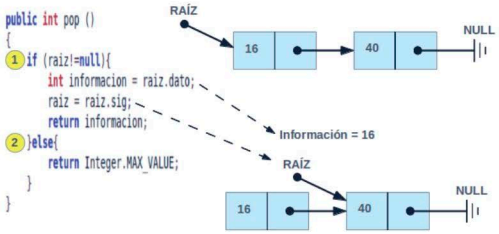


Figura 3.10. *Función pop*

En la imagen anterior se puede observar cómo trabaja la función `pop`. En el primero de los casos, que es el más complejo, se puede ver que el objetivo simplemente es extraer el dato y hacer que la raíz apunte al siguiente nodo al que apunta actualmente.

Vistos ya los procedimientos principales, se muestra el resto del código del programa. Se omite el código de las funciones `push` y `pop` para hacerlo más corto y comprensible:

```
public class Pila {
class Nodo {
int dato;
Nodo sig;
}
private Nodo raiz;
public Pila () {
raiz=null;
}

public void push(int d) {
     .....
}

public int pop ()
{
.....
}

public void print() {
Nodo aux=raiz;
System.out.println("Contenido de la pila:");
while (aux!=null) {
System.out.print(aux.dato+"->");
aux=aux.sig;
}
System.out.println();
}
```

```
public static void main(String[] ar) {
Pila pila1=new Pila();
pila1.push(40);
pila1.push(16);
pila1.push(25);
pila1.print();
System.out.println("Extraemos de la pila:"+pila1.pop());
pila1.print();
}
}
```

EJERCICIO PROPUESTO

Como ejercicio te proponemos que copies el programa anterior, lo compiles, depures y ejecutes para ver cómo funciona.

3.2.2 COLAS

Una vez vistas las pilas va a ser más fácil entender el concepto e implementación de las colas. Una cola dinámica es una estructura FIFO (*First In First Out*), donde el primer dato en entrar es el primero en salir (recuerda que en las pilas el último en entrar es el primero en salir). Nosotros en la vida diaria tenemos que hacer cola en muchos sitios, como la panadería, el estadio de fútbol, el supermercado, etc. En todas esas colas, el primero en llegar debería ser el primero en ser atendido.

Veamos paso a paso cómo se insertan los datos `40`, `16` y `25` en una cola dinámica. Fíjate en que hay dos punteros: `primero`, que apunta al primer elemento de la lista, y `último`, que apuntará al último.

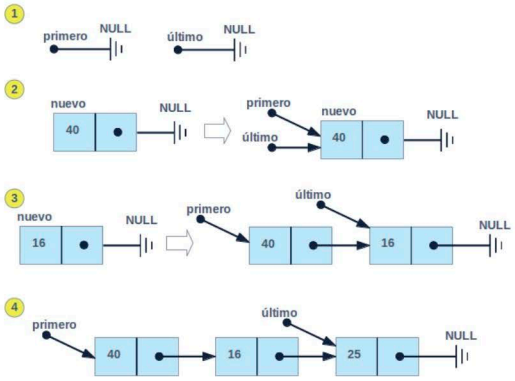


Figura 3.11. Proceso de inserción de elementos en una cola

Como puedes ver, en el paso 1 tenemos dos punteros: uno que se llama `primero` y otro `último`, que apuntan a `null` puesto que no existen datos en la estructura.

Con la inserción de un nuevo nodo (con el dato `40`), `primero` y `último` apuntarán al mismo nodo (puesto que el nodo es el primero y último a la vez).

A partir de este suceso, siempre vamos a insertar los nuevos nodos justo al final (para eso tenemos el puntero que apunta al último). Y si tenemos que extraer algún nodo de la estructura, lo haremos por el primero.

Veamos la implementación de una cola en Java:

```
class Nodo {
int dato;
Nodo sig;
}

private Nodo primero;
private Nodo ultimo;

public Cola () {
primero = ultimo = null;
}
```

Como se puede ver, la implementación del nodo es exactamente igual que para la estructura de una pila. A diferencia de la pila, en la que solamente definimos el puntero raíz, en una cola necesitamos dos punteros (`primero` y `último`).

```
boolean estavacia(){
if (primero == null){
return true;
}else{
return false;
}
}
```

También necesitamos una función auxiliar `estavacia()`, la cual nos va a informar de si la cola está vacía o no. Si la cola está vacía, la variable primero apuntaría a `null`.

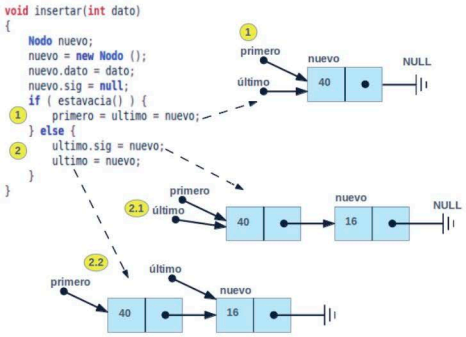


Figura 3.12. Función insertar de la cola

En esta función de inserción (al igual que en el caso de las pilas) se pueden distinguir dos casos. El primer caso es cuando la lista está vacía. Lo que haremos es que los punteros primero y último apunten al nuevo nodo (`primero = ultimo = nuevo`). En el segundo caso lo que hacemos es insertar el nuevo nodo en la última posición (`ultimo.sig = nuevo`), después hacemos que el puntero último apunte al nuevo nodo (que ahora será el `último - ultimo = nuevo`).

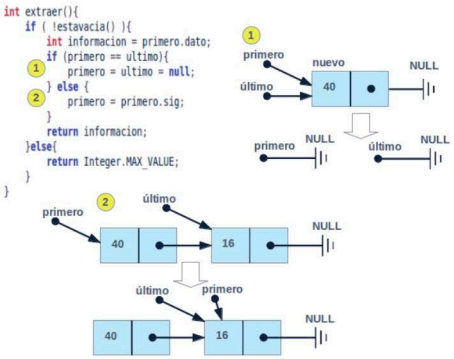


Figura 3.13. *Función extraer de la cola*

Para extraer datos de la lista, el proceso es el siguiente:

- Primero la lista debe contener al menos un elemento (si la cola está vacía, no habrá elementos que extraer). Si la lista contiene al menos un nodo, extraemos la información (`informacion=primero.dato`).
- En caso de que no esté vacía, la cola puede contener solamente un elemento (`primero == ultimo`) y hacemos que el primero y el último apunten a `null` (vaciamos la cola).
- En caso de que tenga dos o más nodos, haremos que el primero apunte al siguiente en la lista (`primero = primero.sig`).

A continuación se muestra el programa completo. Obviamos introducir el código de las funciones estudiadas `estavacia`, `insertar` y `extraer` para que el resto del programa sea más comprensible:

```
public class Cola {

 class Nodo {
 int dato;
 Nodo sig;
 }

 private Nodo primero;
 private Nodo ultimo;
```

```
public Cola () {
primero = ultimo = null;
}

boolean estavacia(){
…..
}

public void insertar(int dato)
{
…..
}

public int extraer(){
…..
}

public void print(){
Nodo aux=primero;
System.out.println("Contenido de la Cola:");
while (aux!=null) {
System.out.print(aux.dato+"<-");
aux=aux.sig;
}
System.out.println();
}

public static void main(String[] ar) {
Cola cola1=new Cola();
cola1.insertar(40);
cola1.insertar(16);
cola1.insertar(25);
cola1.print();
System.out.println("Extraemos de la cola:"+cola1.extraer());
cola1.print();
}
}
```

3.2.3 DATOS ESTRUCTURADOS

Ya hemos visto cómo Java es un lenguaje ideal para implementar tipos abstractos de datos. Hemos visto cómo se implementan tanto la información como las operaciones que hay que realizar sobre la misma en una clase. Las clases nos sirven para abstraer las propiedades del objeto e incluso su implementación. Una clase nos permite encapsular información y operaciones en un mismo fichero.

Usaremos objetos de las clases `pila` y `cola` de los apartados anteriores en nuestros programas y no nos importará lo más mínimo su implementación, lo importante es su interfaz. La interfaz nos ofrecerá las operaciones que podemos realizar con el objeto (su uso) y eso es todo lo que nos hace falta.

EN RESUMEN

En Java podemos implementar tipos abstractos de datos con clases y objetos. La clase es el tipo y los objetos serán instancias de ese tipo.

Se aconseja utilizar los atributos **`private`** para ocultar y **`public`** para hacer accesible desde otros objetos en las clases.

Por ejemplo, en las colas los atributos `primero` y `último` deberían ser privados, e `insertar` y `extraer`, obviamente, públicos, para hacerlos accesibles desde otros objetos.

Java, además de los tipos de datos simples y las clases, puede soportar *frameworks*, como el de **`Collections`**, con los cuales se pueden implementar muchos tipos de datos avanzados. Este *framework* fue incorporado a Java en la versión 1.2 por un gurú de Java llamado Joshua Bloch.

LOS FRAMEWORKS

Un *framework* es un conjunto de interfaces y clases concretas con las cuales puedes crear una aplicación. Utilizando un *framework,* el desarrollo de aplicaciones adquiere una velocidad superior y el número de errores desciende.

A continuación se muestra una imagen de interfaces y clases concretas del *framework* `Collections`:

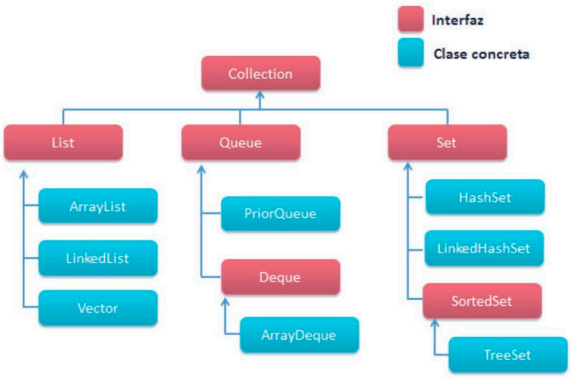


Figura 3.14. *Estructura del framework collection*

3.3 FICHEROS/ARCHIVOS

Cuando queremos que los datos con los que estamos trabajando perduren en el tiempo necesitamos almacenarlos en la memoria secundaria (disco duro, *pendrive*, tarjeta de memoria, etc.). Los ficheros contienen información en modo texto o binario. La información en modo texto puede leerse desde cualquier editor de texto, mientras que con la información en binario, aunque pueda ser accedida, hay que conocer la estructura interna del fichero para poder interpretar los datos que contiene.

El método de acceso a los archivos generalmente difiere dependiendo de la organización interna del mismo. Las formas de acceder a los datos son las siguientes:

- **Acceso secuencial**. El acceso al registro `n` implica la lectura previa de los registros del `1` al `n-1`.
- **Acceso directo**. Los registros se acceden expresando su dirección en el fichero.
- **Acceso por índice**. El acceso a los datos se hace mediante una clave. La clave se busca en una tabla, la cual tiene asociados clave y dirección relativa de los registros. Una vez que se conoce la dirección relativa se accede a los datos. El acceso a los datos, como se puede observar, se hace de forma indirecta.
- **Acceso dinámico**. Se puede acceder a los datos mediante cualquiera de las formas anteriormente citadas.

La escritura y lectura secuencial (que es la más utilizada) de un fichero se puede realizar en Java utilizando las clases `FileInputStream` y `FileOutputStream`.

3.4 MECANISMOS DE GESTIÓN DE MEMORIA

Una de las ventajas de Java es que tiene un mecanismo implícito de gestión de memoria. A diferencia de C++ —en el que es el programador el que tiene que desasignar el espacio concedido a los objetos—, en Java esta tarea la realiza el recolector de basura, con lo cual nos evitamos muchos errores que en algunas ocasiones son difíciles de detectar y corregir.

3.4.1 GESTIÓN AUTOMÁTICA DE MEMORIA EN JAVA

La gestión de memoria en Java la realiza el *Garbage Collector*. El *Garbage Collector*, `gc` o recolector de basura, se ejecuta en segundo plano en un subproceso paralelo a la propia aplicación.

LLAMAR AL GARBAGE COLLECTOR

Si se quiere, se puede hacer una llamada al recolector de basura ejecutando el método `gc()` de la clase `System`. No obstante, no hace falta llamarle pues él se ejecutará cuando lo crea oportuno.

Cuando un programa se ejecuta, el espacio de memoria que el sistema operativo le asigna se va a ir rellenando con objetos. Algunos de estos objetos se destruirán cuando no se necesiten y por lo tanto la memoria se irá llenando de huecos debido a los objetos que dejan de utilizarse. Es por eso que el recolector de basura deberá:

- **Compactar** los huecos de memoria que quedan libres, de forma que quede más espacio para seguir asignando a nuevos objetos.
- **Liberar** los espacios de memoria asignados que ya no se utilizan.

3.4.2 LOS CONSTRUCTORES

Java, al igual que hace con las variables, cuando va a crear un objeto, lo que hace es reservar espacio en la memoria para dicho objeto. En esta fase de construcción del objeto, Java crea un constructor público por defecto del objeto. No obstante, si el programador lo cree oportuno, se puede un constructor diferente que satisfaga las necesidades de la clase.

RECUERDA

El constructor se llama de forma automática siempre que se crea un objeto de una clase.

Por lo tanto, tenemos dos tipos de constructores:

- **Constructor por defecto**. Cuando no se especifica en el código. Se ejecuta siempre de manera automática e inicializa el objeto con los valores especificados o predeterminados del sistema.
- **Constructor definido**. Puede ser más de uno. Tiene el mismo nombre de la clase. Nunca devuelve un valor y no puede ser declarado como `static`, `final`, `native`, `abstract` o `synchronized`. Por regla general se declaran los constructores como públicos (`public`) para que puedan ser utilizados por cualquier otra clase.

RECUERDA

Cuando existe más de un constructor para una clase, se dice que este está sobrecargado.

¿Qué hace Java cuando tiene que cargar una clase?

1. Antes de crear el primer objeto, Java localiza el fichero de la clase en disco (recuerda el fichero `.class`) y lo carga en memoria.
2. Se ejecutarán los inicializadores `static` de la clase (se explican a fondo en la sección final de este apartado).
3. Se crea el objeto.

¿Qué hace Java cuando se crea un objeto?

1 Crea memoria para el objeto mediante el operador `new`.

2 Inicializa los atributos del objeto (solamente los que no fueron inicializados).

3 Se ejecutan los inicializadores de objeto.

4 Llama al constructor adecuado.

Inicialización predeterminada del sistema	
Atributos numéricos	0
Atributos alfanuméricos	Nulo o cadena vacía en caso de `String`
Referencias a objetos	`Null`

Tabla 3.1. *Inicialización predeterminada de variables*

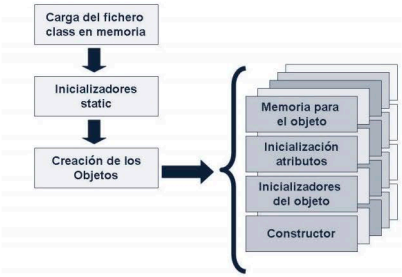


Figura 3.15. *Pasos en la creación de los objetos*

3.4.2.1 Sobrecarga del constructor

¿Cuándo necesitamos sobrecargar o definir múltiples constructores para una clase?

Se definen múltiples constructores para una clase cuando el objeto pueda ser inicializado de múltiples formas. Al sobrecargar un constructor variaremos el tipo y número de parámetros que recibe.

La siguiente clase muestra un ejemplo de sobrecarga de constructores:

```
public class rectangulo
{
    private int ancho;
    private int alto;
    rectangulo(int an, int al){
          this.ancho = an;
          this.alto = al;
    }
    rectangulo(){
          ancho=alto=0;
    }
    rectangulo(int dato){
          ancho=alto=dato;
    }
.........
}
```

RECUERDA

Cuando existe más de un constructor para una clase se dice que este está sobrecargado. Cuando creamos un objeto con `new`, Java elige el constructor más adecuado dependiendo de los parámetros utilizados.

Como según el ejemplo de la clase `rectángulo` tenemos tres constructores, la creación de cada uno de los objetos siguientes se realizará con un constructor diferente:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo();
rectangulo r3 = new rectangulo(8);
```

3.4.2.2 Asignación de objetos

IMPORTANTE

Cuando trabajamos con objetos estamos trabajando con referencias. Una referencia es una localización de la memoria donde se encuentra el objeto.

Como ya se ha dicho, hay que tener en cuenta la utilización de estas referencias cuando se trabaja con objetos. Con un ejemplo muy sencillo se va a comprender qué es este enigma de las referencias. Imaginemos que tenemos la siguiente clase `rectángulo`:

```
public class rectangulo
{
    private int ancho;
    private int alto;
    rectangulo(int an, int al){
          this.ancho = an;
          this.alto = al;
    }
    rectangulo(){ ancho=alto=0;       }
    rectangulo(int dato){ ancho=alto=dato; }
    public int getAncho(){return this.ancho;}
    public int getAlto(){return this.alto;}
    public rectangulo incrementarAncho(){
          ancho++;
          return this;
    }
    public rectangulo incrementarAlto(){
          this.alto++;
          return this;
    }
}
```

Como esta clase es pública, desde el método `main` de otra clase ejecuto el siguiente código:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo();
r2=r1;
r2.incrementarAncho();
r2.incrementarAlto();
System.out.println("Alto: "+r1.getAlto());
System.out.println("Ancho: "+r1.getAncho());
```

La pregunta es la siguiente: ¿qué mostrará el programa por pantalla? Tenemos dos opciones:

Opción 1	Opción 2
Alto: 7	Alto: 8
Ancho: 5	Ancho: 6

La respuesta es la opción 2. Esto es así porque cuando se hace `r2=r1` se copia la referencia al objeto y no el contenido de un objeto en otro. En la siguiente figura se puede ver esto de forma gráfica:

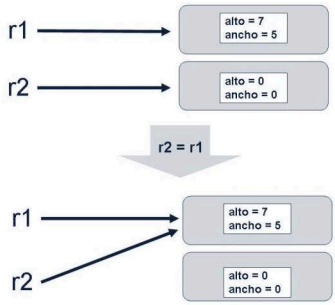


Figura 3.16. *Asignación de referencias*

Entonces, ¿cómo se copia el contenido de un objeto a otro?

Una solución sencilla es emplear un constructor de copia. En la siguiente sección aprenderás a utilizarlo.

3.4.2.3 Constructor copia

Con un constructor de copia se inicializa un objeto asignándole los valores de otro objeto diferente de la misma clase. Este constructor de copia tendrá solo un parámetro: un objeto de la misma clase.

Un constructor de copia para nuestra anterior clase `rectángulo` sería el siguiente:

```
rectangulo (rectangulo r){
       this.ancho = r.getAncho();
       this.alto = r.getAlto();
}
```

En el caso de que tenga este nuevo constructor, puedo hacer uso del mismo cuando cree un objeto `r2` de la misma clase:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(r1);
r2.incrementarAncho();
r2.incrementarAlto();
System.out.println("Alto: "+r1.getAlto());
System.out.println("Ancho: "+r1.getAncho());
```

La utilización del constructor copia "copiará" los miembros del objeto `r1` al objeto `r2`. El programa anterior ahora mostrará por pantalla lo siguiente:

Alto: 7

Ancho: 5

3.4.2.4 Los inicializadores static

Los inicializadores `static` son un bloque de código que se ejecutará una vez solamente cuando se utilice la clase.

Importante. A diferencia del constructor, que se llama cada vez que se crea un objeto de dicha clase, el inicializador solamente se ejecuta la primera vez que se utiliza la clase.

Los inicializadores `static` siguen las siguientes reglas:

- No devuelven ningún valor.
- Son métodos sin nombre.
- Son ideales para inicializar objetos o elementos complicados.
- Permiten gestionar excepciones con `try…catch`.
- Se puede crear más de un inicializador `static` y se ejecutarán según el orden en el que se han definido.
- Se pueden utilizar para invocar métodos nativos o inicializar variables `static`.
- A partir de Java 1.1 existen los inicializadores de objeto, son utilizados en las clases anónimas y no tienen el modificador `static`.

Un ejemplo de clase muy sencillo con varios inicializadores es el siguiente:

```
public class testInicializador{
    static{
          System.out.println("Llamada al inicializador");
    }
    static{
          System.out.println("Llamada al segundo inicializador");
    }
    testInicializador(){
          System.out.println("Llamada al constructor");
    }
}
```

Desde el siguiente programa vamos a crear tres objetos de la clase `testInicializador`:

```
class test {
    public static void main(String[] args) {
          testInicializador t1 = new testInicializador();
          testInicializador t2 = new testInicializador();
          testInicializador t3 = new testInicializador();
 }
}
```

Este programa mostrará por pantalla lo siguiente:

- Llamada al inicializador
- Llamada al segundo inicializador
- Llamada al constructor
- Llamada al constructor
- Llamada al constructor

Presione una tecla para continuar…

3.4.3 LOS DESTRUCTORES

En Java no existen los destructores

Aunque parezca raro empezar un apartado de destructores diciendo que no existen los destructores, en realidad es así. Al contrario que en C++ y otros lenguajes OO, en Java no existen los destructores. En un intento de simplificar las cosas y mejorar la gestión de la memoria, es el propio sistema el que se encarga de eliminar definitivamente los objetos de la memoria cuando le asignamos el valor `null` a la referencia (`referencia = null;`), le asignamos a la referencia un objeto diferente, o bien termina el bloque donde está definida la referencia.

El sistema de liberación de memoria en Java se llama **garbage collector** (recolector de basura), este recolector trabaja de forma automática. Como se vio en apartados anteriores, se le puede sugerir que se active realizando la llamada `System.gc()` pero esta sola no es la única razón para que se active el recolector de basura (se activará cuando él lo decida, generalmente cuando falta memoria).

3.4.3.1 Los finalizadores

Cuando se va a liberar automáticamente la memoria de objetos inservibles, el sistema ejecuta el finalizador de los objetos. El finalizador se caracteriza por no tener valor de retorno ni argumentos, no puede ser `static` y denominarse `finalize()`. Un ejemplo de finalizador es el siguiente:

```
protected void finalize(){System.out.println("Adioossss");}
```

Generalmente, los finalizadores se utilizan para liberar memoria, cerrar ficheros, conexiones, etc. Como no se sabe a ciencia cierta cuándo se van a ejecutar los finalizadores, es el recolector de basura el que se encarga de ello, el consejo es que las operaciones de liberación de ciertos recursos se realicen de forma explícita (a mí no se me ocurriría cerrar una conexión de una base de datos en un finalizador).

Como se verá más adelante, existe una forma de sugerir a Java que ejecute el recolector de basura, y es llamando al método `System.runFinalization()` y luego al recolector de basura `System.gc()`.

Un ejemplo de la llamada al finalizador es el siguiente:

```
public class rectangulo
{
.........
    protected void finalize(){System.out.println("Adioossss");}
.........
}
class testfinalize {

    public static void main(String[] args) {
           for (int i = 0; i < 20; i++) {
            rectangulo r = new rectangulo(5,5);
}

           System.runFinalization();
           System.gc();

    }
}
```

Como se puede observar en el código se ha definido el método `finalize()` como `protected` para evitar que pueda ser invocado desde fuera de la clase.

En el código lo que se ha hecho es crear una serie de objetos cuyo *scope* o ámbito está reducido a un bucle `for`. Una vez realizado esto hay que tener en cuenta que `finalize()` no se invoca cuando termina su *scope* (cuando termina el bucle). Este método se ejecutará justo antes de ejecutarse el garbage collector, por lo que en nuestro código hemos tenido que forzar este hecho con las siguientes líneas:

```
System.runFinalization();
System.gc();
```

Sin las anteriores líneas, el programa no mostrará nada por pantalla.

En resumen:

- El método `finalize()` no es el destructor de C++. En Java no existe el destructor, existe la recolección de basura.
- El método `finalize()` tiene que estar asociado a recuperar la memoria que ha sido utilizada y ya no sirve, no hay que programar otro tipo de cosas aquí.
- Es el sistema y no el programador el que decide cuándo se ejecuta el recolector de basura.
- Los objetos pueden o no ser eliminados por el recolector de basura. En algunos casos no se eliminan si no existe una necesidad de memoria.
- Generalmente, se utiliza `finalize()` cuando se hacen llamadas a métodos nativos (por ejemplo en C o C++) para reservar memoria y luego esta necesita ser liberada.
- Visto lo anterior, es normal pensar que el método `finalize()` no se va a utilizar mucho en Java y, salvo necesidad específica, esto es siempre así.

3.5 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El método `compareTo` distingue mayúsculas de minúsculas.
b) Java, además de los tipos de datos simples y las clases, puede soportar *frameworks*.
c) Las pilas y las colas suelen ser estructuras dinámicas, pero también se pueden representar mediante estructuras estáticas.
d) La gestión de memoria en Java la realiza el Garbage Collector.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El método `equals` permite discernir si dos referencias son el mismo objeto.
b) La línea `cad1=cad1.concat(" Rosique");` dará fallo de compilación puesto que el método `concat` no existe.
c) El método `clone` permite duplicar un objeto.
d) Las estructuras dinámicas en Java generalmente se implementan mediante un conjunto de nodos.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un *framework* es un conjunto de interfaces y clases concretas con las cuales puedes crear una aplicación.
b) Las cadenas de caracteres en Java se tratan como objetos de la clase `String`.
c) La clase `String` pertenece al paquete `java.System`.
d) En Java podemos implementar tipos abstractos de datos con clases y objetos.

4 Relaciones entre clases

Comprender las relaciones entre clases es comprender la esencia y la necesidad de la herencia. En este capítulo se profundiza en ambos conceptos, los cuales son un pilar básico en la POO.

4.1 TIPOS DE RELACIONES ENTRE CLASES

Las clases se relacionan unas con otras de diferente manera y dependiendo de la relación que haya entre ellas la comunicación entre los objetos de las mismas será diferente. Los mensajes, por así decirlo, van a navegar por las relaciones existentes entre las distintas clases.

En este apartado vamos a ver tres tipos distintos de relaciones:

- Agregación/composición.
- Generalización/especialización.
- Asociación.

4.1.1 AGREGACIÓN/COMPOSICIÓN

Un caso de relaciones entre clases es el de agregación y composición. En estos tipos de asociaciones se relaciona el todo con cada una de sus partes.

Este tipo de asociaciones se muestra gráficamente con un rombo en uno de los extremos de la relación.

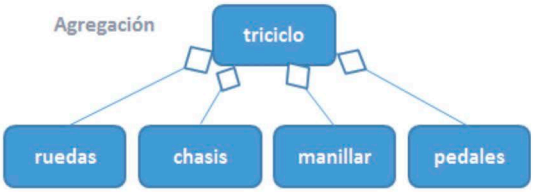


Figura 4.1. *Relación de agregación entre varias clases*

Por ejemplo, en las relaciones de **agregación** el rombo del extremo tendrá fondo blanco mientras que en las relaciones de **composición** el rombo tendrá fondo negro.

En una relación de agregación los agregados o componentes podrían formar parte de cualquier otra relación mientras que en una relación de composición su existencia es posible únicamente por la propia relación.

Figura 4.2. *Relación de composición entre varias clases*

Por ejemplo, en las dos relaciones anteriores, la clase `triciclo` estará compuesta de cuatro clases (*ruedas*, *chasis*, *manillar* y *pedales*). Cualquiera de estas clases podría formar parte de otra relación, como bicicleta, o podría tener sentido por sí misma, como las ruedas. Sin embargo, en la relación de composición de la ventana, la cabecera no tiene sentido de existencia por sí misma. La cabecera existe porque existe la ventana.

4.1.2 GENERALIZACIÓN/ESPECIALIZACIÓN

Las relaciones de **generalización** y **especialización** son las típicas relaciones de herencia en Java. Es la relación que existe entre una clase(superclase) y sus subclases. Los objetos de cada una de estas clases tendrán comportamiento y atributos similares.

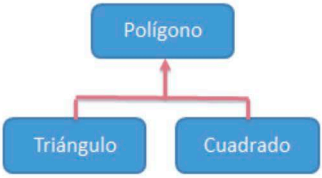


Figura 4.3. *Relación sencilla de generalización/especialización*

Su implementación en Java será algo parecido a esto:

```
public class poligono{
...
}

public class triangulo extends poligono{
...
}

public class cuadrado extends poligono{
...
}
```

La relación entre este tipo de clases se expresaría con las palabras "es un". Por ejemplo, un triángulo es un polígono y un cuadrado es un polígono también. Luego las propiedades de un polígono (área, altura, etc.) serán propiedades también de un cuadrado y un triángulo; sin embargo, algunas propiedades del triángulo no serán compartidas por el cuadrado, y viceversa.

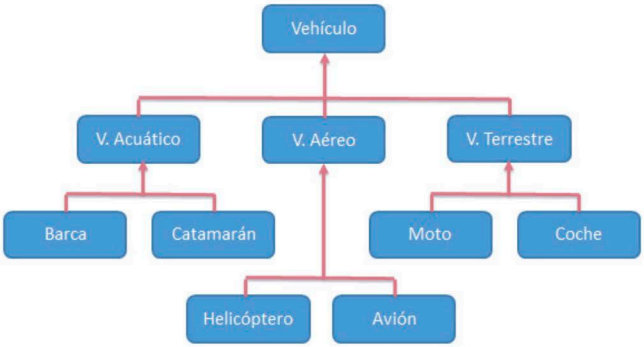


Figura 4.4. *Jerarquía de clases*

Generalmente las clases se organizan en estructuras jerárquicas, que forman una **taxonomía**. Las subclases van heredando de las clases padres de las cuales derivan y van añadiendo nuevas características que hacen que se diferencien unas de otras.

Cuando diseñes un diagrama de clases, coloca los atributos y los métodos en el nivel más alto de la jerarquía que sea aplicable.

4.1.3 ASOCIACIÓN

Las asociaciones son relaciones entre objetos que implican una conexión entre ellos. Cuando se quiere representar una relación de asociación se hace mediante una línea continua que une las clases asociadas.

Figura 4.5. *Relación de asociación*

Generalmente, las asociaciones son bidireccionales. Conociendo el nombre de un equipo podemos saber los jugadores que juegan en él; y viceversa, si conocemos el nombre de un jugador podemos deducir el nombre del equipo donde juega. La relación se puede recorrer en ambos sentidos.

En ocasiones no queremos que la relación se recorra en ambos sentidos y por tanto da lugar a una relación de asociación unidireccional y para reflejar este hecho colocamos una flecha de un solo sentido.

Figura 4.6. *Relación de asociación unidireccional*

Un ejemplo de una relación de asociación unidireccional se expresaría en Java de la siguiente manera:

```
class persona{
    fecha fnac;
    void setFechaNacimiento(fecha f){
          fnac=f;
    }
    fecha getFechaNacimiento(){
          return fnac;
    }
```

En el ejemplo anterior hemos tomado la clase `fecha` como un tipo de dato con el cual podemos realizar operaciones relacionadas con la fecha. Se ha creado una clase `persona` que tiene como fecha de nacimiento un atributo de ese tipo y dos métodos (*getter* y *setter*) para establecer y recuperar dicha fecha.

4.2 HERENCIA

La herencia es el mecanismo que utiliza la POO para la reutilización de las clases. Extiende la funcionalidad de una clase sin tener que reescribir código. Las subclases poseerán atributos y métodos que no existen en las superclases.

4.2.1 LA HERENCIA EN JAVA

La herencia es la base de la reutilización del código. Cuando una clase deriva de una clase padre, esta hereda todos los miembros y métodos de su antecesor. También es posible redefinir (*override*) los miembros para adaptarlos a la nueva clase o ampliarlos. En general, todas las subclases no solo adoptan las variables y comportamiento de las superclases, sino que los amplían.

RECUERDA

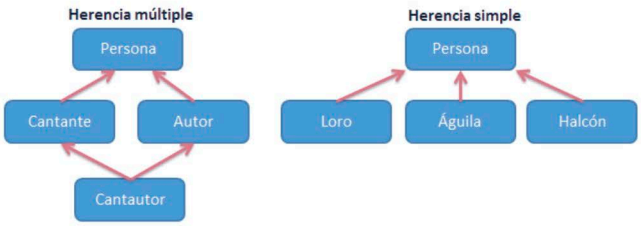


Figura 4.7. *Ejemplo de herencia múltiple*

En Java, al contrario que en C++, no se permite la herencia múltiple. Es decir, una clase no puede heredar de varias clases, lo que hace que las relaciones entre ellas sean más sencillas y menos complejas.

En la siguiente figura se muestra un ejemplo de herencia:

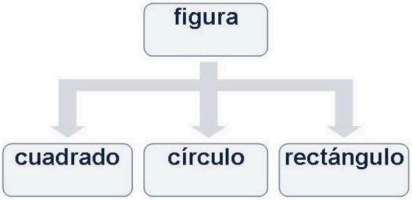


Figura 4.8. *Ejemplo de herencia*

Como se puede ver en la figura, en este árbol de herencia tendremos la clase `figura`, de la que heredan las clases `cuadrado`, `círculo` y `rectángulo`. Como es obvio, `cuadrado`, `círculo` y `rectángulo` tienen una característica en común y es que todas son figuras. Otra característica que presenta este árbol es que, en este caso, las figuras por sí mismas no existen; es decir, existirán pero siempre deberá ser a través de una clase de nivel inferior (`cuadrado`, `círculo` o `rectángulo`).

Para indicar que una clase hereda de otra se etiqueta con la cláusula **`extends`** detrás del nombre de la clase. Por ejemplo, para indicar que la clase `rectángulo` hereda de la clase `figura` escribiremos lo siguiente:

```
class rectangulo extends figura { ... }
```

Igual podremos hacer para las demás clases:

```
class circulo extends figura { ... }
class cuadrado extends figura { ... }
```

RECUERDA

Todas las clases tienen una superclase o clase padre. Cuando escribas una clase, si esta no hereda de ninguna clase concreta siempre heredará de la clase `Object` (`java.lang.Object`).

Imaginemos que queremos realizar una estructura de clases como la que se muestra a continuación. El código de las clases `figura` y `cuadrado` se muestra junto a la siguiente imagen:

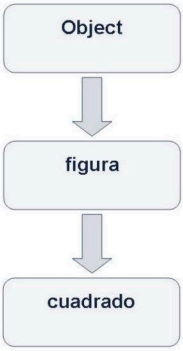


Figura 4.9. *Herencia entre varias clases*

```
public class figura{
 String color;
 public void setColor(String s){color=s;}
 public String getColor(){return color;}
}
public class cuadrado extends figura{
 private int lado;
 cuadrado(int l){ this.lado = l; }
 public int getArea(){ return lado*lado; }
}
```

Al utilizar la cláusula `extends` indicamos lo siguiente:

- La clase `cuadrado` es una subclase de la clase `figura`.
- La clase `cuadrado` puede utilizar los métodos de la clase `figura` aunque no estén declarados en la clase `cuadrado` (siempre y cuando no estén como `private` en la clase `figura`).
- Obviamente, los métodos de la subclase no pueden ser utilizados en la superclase o clase principal.

RECUERDA

Las clases heredan el comportamiento de sus antecesores (padres) pero no lo heredan de otras subclases (hermanos).

Imaginemos que queremos testear el comportamiento de la jerarquía anterior. Para ello crearemos la siguiente clase:

```
class testFiguras {
    public static void main(String[] args) {
           cuadrado c=new cuadrado(5);
           c.setColor("Verde");
           System.out.println(c.getColor());
           System.out.println(c.getArea());
 }
}
```

En esta clase se puede observar cómo se llama a métodos de la superclase `figura` y de la subclase `cuadrado`. Solamente hemos tenido que crear una clase `cuadrado`, puesto que los atributos y métodos de la clase `figura` los ha heredado la clase `cuadrado`.

4.2.2 LA CLASE OBJECT

IMPORTANTE

La clase `object` es la raíz jerárquica de Java.

Cualquier clase implementada en Java siempre va a ser una subclase de la clase `object`. Eso quiere decir que va a heredar todos los métodos de `object`. De todos los métodos de la clase object vamos a ver con más profundidad los siguientes:

Método	Descripción
`clone()`	Permite "clonar" un objeto.
`equals()`	Permite comparar un objeto con otro.
`toString()`	Devuelve el nombre de la clase.
`finalize()`	Método invocado por el recolector de basura (*garbage collector*) para borrar definitivamente el objeto.

Tabla 4.1. *Métodos de la clase Object*

Método `clone()`

El método `clone` nos permite copiar un objeto en otro. Utilizar este método equivaldría a utilizar un constructor de copia. La clase base `object` tiene el método `clone()`, que es el mecanismo que utiliza Java para clonar objetos. Es posible y en muchos casos necesario implementar un método `clone`, el cual sobreescribirá al método `clone` de su superclase y podrá actuar de una forma más específica que el método genérico `clone()`.

El método genérico `clone()` hace una copia superficial del objeto.

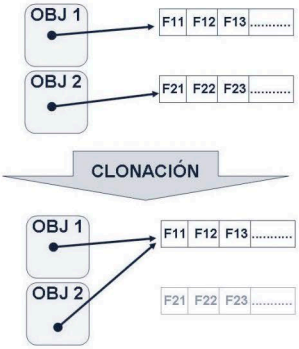


Figura 4.10. *La copia superficial*

Como se puede ver en la figura anterior, la copia superficial únicamente hace una copia del contenido de un objeto en otro, lo que en algunas ocasiones provoca que la modificación del contenido de un objeto implique el cambio en el clonado, y viceversa.

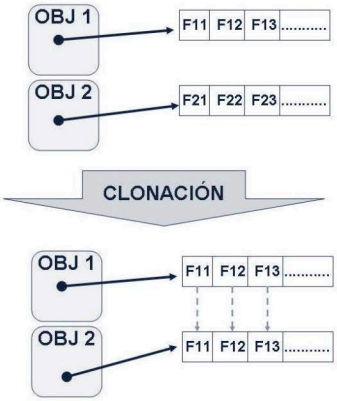


Figura 4.11. *Copia en profundidad*

Por el contrario, las copias en profundidad pueden hacer una copia selectiva del contenido de un objeto en otro. En este caso ambos objetos vivirán "vidas independientes".

Un ejemplo de realizar una clonación de un objeto en Java sería el siguiente:

```
public class rectangulo implements Cloneable
{
    private int ancho;
    private int alto;
    private String nombre;
    public Object clone(){
 Object objeto=null;
 try{
 objeto =super.clone();
 }catch(CloneNotSupportedException ex){
 System.out.println(" Error al duplicar");
 }
 return objeto;
    }
........
}
```

```
class testeoclone {

    public static void main(String[] args) {
          rectangulo r1 = new rectangulo(5,7);
          rectangulo r2 = (rectangulo) r1.clone();
          r2.incrementarAncho();
          r2.incrementarAlto();
          r1.setNombre("Chiquito");
          r2.setNombre("Grande");
          System.out.println("Alto: "+r1.getAlto());
          System.out.println("Ancho: "+r1.getAncho());
          System.out.println("Alto: "+r2.getAlto());
          System.out.println("Ancho: "+r2.getAncho());
          System.out.println("Nombre: "+r1.getNombre());
          System.out.println("Nombre: "+r2.getNombre());
 }
}
```

Como se puede observar, la clase objeto de la clonación deberá implementar la interfaz cloneable. Si no se implementa esta interfaz, el programa lanzará una excepción del tipo `CloneNotSupportedException`. También se ha implementado el método `clone()`, el cual hace una llamada al método `clone()` de su clase base.

IMPORTANTE

Muchas veces es más cómodo para el programador utilizar el constructor de copia que el método `clone()`.

Método `equals()`

El método `equals` permite realizar una comparación entre un objeto y otro. Lo que hace es comprobar que ambas referencias sean iguales, con lo cual no obtenemos más ventaja que con el operador = =. No hace una comparación en profundidad sino que se limita a comprobar las referencias de los objetos. Si se quiere realizar una comprobación en profundidad habrá que reescribir este método.

Un ejemplo de utilización de este método es el siguiente:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(5,7);
rectangulo r3 = r1;
if (r1.equals(r2)){
      System.out.println("Iguales r1 y r2(equals)");
}
if (r1.equals(r3)){
      System.out.println("Iguales r1 y r3(equals)");
}
```

El resultado en pantalla de ejecutar el código anterior será: `"Iguales r1 y r3(equals)"`. Para que la primera comprobación sea verdadera habrá que reescribir el método `equals`.

Método `toString()`

El método `toString()` permite obtener el nombre de la clase desde el cual fue invocado. Además del nombre de la clase, devuelve el carácter `'@'` y la representación hexadecimal del código *hash* del objeto. Un ejemplo de la llamada a este método es el siguiente:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(5,7);
rectangulo r3 = r1;
System.out.println(r1.toString());
System.out.println(r2.toString());
System.out.println(r3.toString());
```

Este código devolverá por pantalla lo siguiente:

```
rectangulo@19821f
rectangulo@addbf1
rectangulo@19821f
```

Como podemos observar en el código, al hacer `r3=r1` lo que hacemos es que ambas referencias apunten al mismo objeto, con lo cual al invocar al método `toString()` el resultado será el mismo.

Método `finalize()`

Cuando el recolector de basura de Java (*garbage collector*) tiene constancia de que no existen más referencias a un objeto concreto, invoca a este método y se encarga de liberar su memoria ocupada. Si el programador necesita realizar una acción una vez destruido un objeto, deberá reescribir este método.

4.2.3 LAS INTERFACES

Como ya sabemos, un objeto interactúa con el mundo exterior a través de su **interfaz**. En el caso de un ordenador, por ejemplo, las interfaces con el mundo exterior serán la pantalla, el ratón, el teclado, etc. Cuando nosotros tecleamos en un ordenador, las teclas o la pantalla sirven para comunicarnos con la parte interna del equipo. Imaginemos que actualizamos la memoria, el procesador y la placa base del equipo conservando la parte software del mismo. La interfaz será exactamente la misma y las personas interactuarán exactamente igual con ella. Lo único que notarán es una mejora del rendimiento. Con los objetos pasa exactamente lo mismo, la interacción con el mundo exterior es a través de sus métodos. Los métodos componen la interfaz del objeto con el mundo exterior.

Una interfaz (o *interface*) es un grupo de métodos con sus cuerpos vacíos. Por ejemplo, la interfaz `figura (intfigura)` podría ser la siguiente:

```
public interface intfigura{
    int area();
}
```

La interfaz define el método `área`, para su posterior desarrollo en las clases que implementen esta interfaz. Una de las clases que podría implementar esta interfaz es la clase `rectángulo`:

```
public class rectangulo implements intfigura{
    private int ancho;
    private int alto;
    rectangulo (int an, int al){
           this.ancho = an;
           this.alto = al;
    }

    public int area(){ return ancho*alto; }
}
```

Como se puede observar en la declaración, la clase `rectángulo` implementa la interfaz `intfigura`.

RECUERDA

Para compilar correctamente una clase que implementa una interfaz, esta debe contener los métodos declarados en dicha interfaz.

4.3 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las relaciones de generalización y especialización son las típicas relaciones que existen entre una clase(superclase) y sus subclases.
b) El polimorfismo extiende la funcionalidad de una clase sin tener que reescribir código.
c) Las asociaciones son relaciones entre objetos que implican una conexión entre ellos.
d) En Java, al contrario que en C++, no se permite la herencia múltiple.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las asociaciones de agregación se muestran gráficamente con un rombo en uno de los extremos de la relación.
b) Para compilar correctamente una clase que implementa una interfaz, esta debe contener al menos un método declarado en dicha interfaz.
c) El método `clone` nos permite copiar un objeto en otro.
d) El método `toString()` permite obtener el nombre de la clase desde el cual fue invocado.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Cuando escribas una clase, si esta no hereda de ninguna clase concreta siempre heredará de la clase `Object`.
b) El método `equals` permite realizar una comparación entre un objeto y otro.
c) El método `iguals()` permite realizar una comparación entre un objeto y otro.
d) Generalmente las clases se organizan en estructuras jerárquicas, las cuales forman una taxonomía.

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las asociaciones de agregación se muestran gráficamente con un rombo con fondo blanco en uno de los extremos de la relación.
b) La herencia es la base de la reutilización del código.
c) El método `toString()` permite obtener los atributos de la clase desde la cual fue invocado.
d) Las asociaciones de composición se muestran gráficamente con un rombo con fondo negro en uno de los extremos de la relación.

5 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Una interfaz (*interface*) es un grupo de métodos en el que algunos pueden tener código.
b) Las relaciones de generalización y especialización son las típicas relaciones de herencia en Java.
c) Cualquier clase implementada en Java siempre va a ser una subclase de la clase `object`.
d) La herencia es el mecanismo que utiliza la POO para la reutilización de las clases.

5 Lenguajes de programación orientados a objetos. Polimorfismo, excepciones y librerías de clases

En este tema se verán conceptos avanzados de la programación con Java como es el polimorfismo, las excepciones, las librerías de clases, los hilos y la programación en red.

5.1 LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETOS

A continuación, se citan algunas de las características fundamentales de la programación orientada a objetos:

- **Abstracción**. Según la RAE, abstraer es "separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción". Cuando se programa orientado a objetos, lo que se hace es abstraer las características de los objetos que van a formar parte del programa, y crear las clases con sus atributos y sus métodos.
- **Encapsulamiento**. El encapsulamiento es una de las propiedades fundamentales de la programación orientada a objetos. Cuando se programa orientado a objetos, los objetos se ven según su comportamiento externo. Por ejemplo, en la clase `pájaro` se le puede enviar un mensaje para que cante y el objeto `pájaro` ejecutará su método `cantar`. Lo más interesante de todo esto es que el programador no tiene por qué saber cómo funciona internamente el método `cantar`, simplemente lo utiliza.

RECUERDA

En la POO las clases son vistas como una caja negra. Los programadores no tienen por qué saber nada de los datos que almacenan ni del interior de los métodos que permiten manipularlos, solamente tienen que conocer su interfaz.

- **Herencia**. Todas las clases se estructuran formando jerarquías de clases.

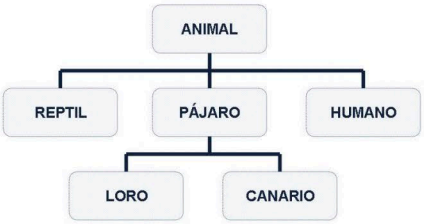


Figura 5.1. *Jerarquía de clases*

Las clases pueden tener superclases (la clase `pájaro` tiene la superclase `animal`) y subclases (la clase `pájaro` tiene las subclases `loro` y `canario`). También existe la posibilidad que una clase herede de varias superclases.

RECUERDA

En Java, una clase SOLO puede tener UNA superclase. Este hecho se denomina "herencia simple". En C++ se permite la herencia múltiple. Java puede simular la herencia múltiple utilizando interfaces.

Cuando una clase hereda de una superclase obtiene los métodos y las propiedades de dicha superclase. Además, la funcionalidad propia de la misma clase se combinará con la heredada de la superclase.

- **Polimorfismo**. El polimorfismo permite crear varias formas del mismo método, de tal manera que un mismo método ofrezca comportamientos diferentes.

5.1.1 LOS LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETOS MÁS HABITUALES EN EL MERCADO

A continuación se presentan algunos de los lenguajes de programación orientados a objetos más utilizados actualmente en el mercado. Existen muchos más, pero hemos intentado citar los más relevantes:

Objetive-C. Es un lenguaje orientado a objetos influido por C y SmallTalk. Apareció en 1980 y fue diseñado por Brad Cox. Su importancia radica en que se usa como lenguaje de programación en XCode, que es la herramienta de Apple para realizar programas para iOS y Mac OS X.

Figura 5.2. *Logo de xCode. Herramienta de programación para Mac*

C++. Es un lenguaje de programación diseñado en los 80 por Bjarne Stroustrup. Se considera lenguaje de programación híbrido o multiparadigma porque en realidad es un lenguaje C que permite la manipulación de objetos y a diferencia de Smalltalk no es un lenguaje orientado a objetos puro.

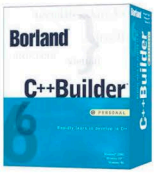


Figura 5.3. *Herramienta Builder C++. Un clásico de la programación visual en C++*

Java. Lenguaje desarrollado por James Gosling, de Sun, apareció en 1995. La última versión es la JSE 8. Es un todoterreno y sus cualidades multiplataforma lo han encumbrado como el lenguaje de Internet. Podemos encontrar Java en dispositivos móviles, sistemas empotrados, navegadores, sistemas servidor, aplicaciones de escritorio, *apps*, etc.

Figura 5.4. *Logo del lenguaje Java*

PHP. Lenguaje multiparadigma con características de orientación a objetos diseñado por Rasmus Lerdorf. Apareció, como Java, en 1995. Es un lenguaje utilizado en en lado del servidor. El código es interpretado por un servidor PHP que junto con un servidor web produce una página HTML que es la que el usuario visualiza. Utilizado en múltiples sistemas web (CMS, *blogs*, etc.). Utilizado como lenguaje base en Wordpress, que es el sistema más utilizado para diseñar *blogs* y sitios web.

Figura 5.5. *Logo del lenguaje PHP*

Python. Diseñado por Guido van Rossum, es un lenguaje multiplataforma como Java. Es un lenguaje interpretado como PHP, por ejemplo, y es administrado por la Python Software Foundation mediante una licencia de código abierto.

Figura 5.6. *Logo del lenguaje Python*

VB.NET. Es la evolución del famoso Visual Basic que se implementó en el *framework* .NET. Muchos de los cambios que se hicieron no eran compatibles con las versiones anteriores de Visual Basic. Si programas en .NET lo normal es que utilices el Microsoft Visual Studio, aunque existen otras alternativas menos potentes pero libres como SharpDevelop o MonoDevelop.

Figura 5.7. *Logo del lenguaje .NET*

5.2 POLIMORFISMO

Según la **RAE**, el polimorfismo es la "cualidad de lo que tiene o puede tener distintas formas". El polimorfismo en programación orientada a objetos permite abstraer y programar de forma general agrupando objetos con características comunes y jerarquizándolos en clases. Como se ha dicho anteriormente, existe una clase que es la clase padre de todas las demás y es la `java.lang.Object`. Cualquier clase creada descenderá de `Object`.

RECUERDA

El polimorfismo se consigue en Java mediante las clases abstractas y las interfaces. Concretamente las interfaces amplían enormemente las posibilidades del polimorfismo.

Un aspecto muy importante del polimorfismo es cuando se crea una referencia a un objeto de una clase base, esa misma referencia puede servir para referenciar a objetos de clases derivadas.

Imaginemos que tenemos este árbol jerárquico:

Figura 5.8. *Relación entre las clases persona, empleado y encargado*

Tenemos la clase `persona`, de la cual desciende la clase `empleado`. La clase `persona` tendrá métodos genéricos que puedan ser utilizados por cualquier persona, como, por ejemplo, establecer y devolver el nombre.

La clase `empleado` tendrá otro tipo de métodos más específicos, como `obtenerSueldo`, el cual devolverá el sueldo base, así como `setSueldobase`, que establecerá el sueldo base del empleado.

Los encargados son personas con responsabilidades en la empresa y sea cual sea su trabajo cobrarán un 10 % más que un empleado normal.

La implementación de la jerarquía anterior será la siguiente:

```
public class persona {
    private String nombre;
    public void setNombre(String nom){
          nombre = nom;
    }
    public String getNombre(){
          return nombre;
    }
}
public class empleado extends persona{
    protected int sueldoBase;
    public int getSueldo(){ return sueldoBase; }
    public void setSueldoBase(int s){ sueldoBase = s; }
}
public class encargado extends empleado{
    public int getSueldo(){
          Double d = new Double(sueldoBase*1.1);
          return d.intValue();
    }
}
```

Imaginemos que realizamos lo siguiente con la clase test:

```
class test {
    public static void main(String[] args) {
          persona p1;
          p1 = new empleado();
          p1.setNombre("Isaac Sanchez");
          p1.setSueldoBase(100); //dará error al compilar

          empleado e1;
          e1 = new encargado();
          e1.setSueldoBase(500);
          e1.setPuesto("Jefe almacen"); //dará error al compilar
          System.out.println(e1.getSueldo());
 }
}
```

Vamos a comentar el código de la clase anterior:

```
          persona p1;
          p1 = new empleado();
          p1.setNombre("Isaac Sanchez");
          p1.setSueldoBase(100); //dará error al compilar
```

En el código anterior creamos una referencia `persona` que apunta a un objeto de la clase `empleado`. La variable `p1` podrá hacer llamadas a métodos de la clase `persona`, pero no de la clase `empleado`; por lo tanto, la llamada al método `setSueldoBase` dará un error de compilación.

```
empleado e1;
e1 = new encargado();
e1.setSueldoBase(500);
e1.setPuesto("Jefe almacen"); //dará error al compilar
System.out.println(e1.getSueldo());
```

Por otra parte, vemos que el código anterior crea la referencia a `empleado` pero apunta a un objeto del tipo `encargado`. La llamada al método `setPuesto` dará error por lo explicado anteriormente, pero no así la llamada al método `getSueldo()`. La pregunta que nos hacemos es la siguiente: ¿el programa mostrará por pantalla 500 o 550?

La solución es 550. Aunque la referencia se creó para la clase `empleado` y solamente se puede llamar a métodos de dicha clase, el método `getSueldo()` está sobreescrito, y como `e1` apunta a un objeto de la clase `encargado`, Java resuelve que tiene que ejecutar el método de dicha clase.

La sobreescritura de métodos se ve con mayor profundidad en el siguiente apartado.

RECUERDA

Cuando la referencia se creó para la clase `base`, solamente se puede hacer llamadas a métodos de dicha clase base.

Vamos a ver con mayor detalle por qué el programa mostró 550 y no 500 por la salida estándar:

En Java existen dos tipos de vinculaciones (con vinculación nos referimos a la llamada realizada a un método y al código que se va a ejecutar en dicha llamada): la vinculación temprana y la vinculación tardía.

- **Vinculación temprana** o en **tiempo de compilación**. Con métodos normales o sobrecargados Java utiliza la vinculación temprana.
- **Vinculación tardía** o en **tiempo de ejecución**. Cuando se redefinen métodos se realizará dicha vinculación (salvo métodos definidos como `final`).

En nuestro caso se ha declarado el método `getSueldo()` en la clase `empleado` (clase padre) y se ha sobreescrito en la clase derivada `encargado` (clase hija). Cuando en tiempo de ejecución se llama a este método, el tipo de objeto al que apunta la variable prima sobre el tipo de la referencia. Es en tiempo de ejecución cuando se comprueba que aunque la referencia es de tipo empleado, la variable `e1` apunta a un objeto de tipo encargado y el método de esta clase es el que se va a ejecutar.

Aunque el polimorfismo confiere muchas ventajas para el lenguaje, hemos descubierto una limitación, y es el tipo de la referencia la que limita los métodos que podemos ejecutar (en nuestro ejemplo la llamada al método `setPuesto` dará error) o las variables miembro accesibles.

RECUERDA

Se puede crear un objeto cuya referencia sea una interfaz. Este objeto solamente podrá ejecutar los métodos de dicha interfaz pero no podrá utilizar los métodos y miembros de dicho objeto. La interfaz concebida así es una forma de unificación de uso entre clases muy diferentes.

Como hemos visto anteriormente, nuestro programa va a dar errores de compilación. Entonces, ¿qué hacer para que el código anterior compile y funcione? La respuesta a esta pregunta es utilizar un *cast* explícito (obligar al compilador a transformar obligatoriamente el objeto en otro). El *cast* entre objetos se verá en profundidad en un apartado posterior; por lo tanto, de momento se incluyen las líneas que dan problema y debajo la línea reprogramada que solventa dicho problema:

```
p1.setSueldoBase(100); //dará error al compilar
((empleado)p1).setSueldoBase(100); //corregida

e1.setPuesto("Jefe almacen"); //dará error al compilar
((encargado)e1).setPuesto("Jefe almacen"); //corregida
```

5.2.1 SOBRECARGA DE MÉTODOS

Una de las propiedades fundamentales de los lenguajes orientados a objetos es la sobreescritura u *overriding* de métodos. Obviamente, los métodos son los únicos que se pueden sobreescribir; con los elementos miembro esta técnica no es posible.

La sobreescritura permite modificar el comportamiento de la clase padre (también llamada clase principal o superclase).

Para que dicho método con diferente funcionalidad sea sobreescrito, deberá cumplir los siguientes preceptos:

- Tiene que tener el mismo nombre (esto es obvio).
- El retorno de la clase padre e hijo deberá ser del mismo tipo.
- Deberá conservar la misma lista de argumentos que el mismo método en la clase padre.

Un ejemplo de sobrecarga de métodos sería el siguiente:

```
package sobreescribe;
public class Pajaro {
protected String nombre;
protected String color;
public String getDetalles(){
 return "Nombre: " + nombre + "\n" + "Color: " + color;
}
}

package sobreescribe;
public class Loro extends Pajaro {
protected String pedigri;
public String getDetalles(){
 return "Nombre: " + nombre + "\n" + "Color: " + color + "\n" + "Pedrigri: "+ pedigri;
}
}
```

Según el ejemplo anterior, se puede observar lo siguiente:

- La clase `Loro` desciende de la clase `Pájaro`.
- La clase `Loro` sobreescribe el método `getDetalles()`, ambas con el mismo nombre.
- El método `getDetalles()` de clase padre e hija tienen la misma lista de argumentos.
- El método `getDetalles()` de clase padre e hija devuelven un objeto `String` (mismo tipo).
- El método `getDetalles()` de clase padre e hija tienen el mismo modificador de acceso (`public`).

5.2.2 SOBREESCRITURA DE MÉTODOS

RECUERDA

La sobrecarga es la implementación varias veces del mismo método con ligeras diferencias adaptadas a las distintas necesidades de dicho método.

Como se ha dicho antes, la sobrecarga implica una implementación repetida del mismo método. Para crear métodos sobrecargados deberemos crear métodos con el mismo nombre pero con distinta lista de parámetros. A continuación se enumeran las reglas para sobrecargar un método:

- Los métodos sobrecargados deben cambiar la lista de argumentos obligatoriamente.
- Un método puede estar sobrecargado en la clase o en una subclase.
- Al sobrecargar un método se pueden utilizar las mismas excepciones o añadir algunas.
- Los métodos sobrecargados pueden cambiar el tipo de retorno o el modificador de acceso.

Imaginemos que tenemos una clase `persona` en la que vamos a almacenar datos de ciertas personas como el nombre, teléfono, dirección, etc. Tenemos un problema y es que vamos a almacenar para su posterior tratamiento el primer y segundo apellido de todos los individuos. Imaginemos que tenemos un inglés o un italiano, que por costumbre no utilizan su segundo apellido. Esta es una buena ocasión de utilizar un método sobrecargado. Un ejemplo de esto es el siguiente:

```
public class persona {
    private int sinsegundo=0;
    private String nombre;
    private String apellido1;
    private String apellido2;
    public void setNombre(String nom,String ape1,String ape2){
           nombre = nom;
           apellido1 = ape1;
           apellido2 = ape2;
    }
    public void setNombre(String nom,String ape1){
           nombre = nom;
           apellido1 = ape1;
           sinsegundo = 1;
    }
}
```

Obsérvese que el código anterior cumple con todas las reglas enumeradas anteriormente.

RECUERDA

Los métodos sobrecargados deben cambiar la lista de argumentos del método.

5.3 PROGRAMACIÓN AVANZADA. EXCEPCIONES

En el capítulo 2 vimos cómo una excepción era un evento producido durante la ejecución del programa que hacía que el flujo normal de las instrucciones del programa se interrumpiera. También vimos cómo era posible manejar dichas excepciones de tal manera que cuando se producía la excepción, esta era tratada en un bloque de código especial (`catch` o `finally`).

RECUERDA

La ventaja de utilizar excepciones es que se independiza el código regular del código de tratamiento de errores haciendo el programa más legible.

Algunos problemas por los que se ha producido una excepción suelen ser datos inválidos introducidos por el usuario (abrir ficheros inexistentes, pédida de la conexión en una comunicación, falta de memoria, etc.). Muchas de esas excepciones son errores de programación y otras no. Java clasifica las excepciones en tres categorías:

- ***Checked exceptions* o excepciones comprobadas**. Son problemas que en muchos casos no pueden ser evitados por el programador, como por ejemplo cuando se lee un fichero y este no existe.
- ***Runtime exceptions* o excepciones en tiempo de ejecución**. A menudo este tipo de excepciones sí podrían ser evitadas por el programador, pero por la razón que sea este no lo ha hecho.
- **Errores**. En principio no deben catalogarse como excepciones puesto que no son culpa ni del programador ni del usuario del programa. Por ejemplo, un error sería si se queda el sistema sin memoria.

5.3.1 EXCEPCIONES DEFINIDAS Y LANZADAS POR EL PROGRAMADOR

Veamos un ejemplo comentado de cómo un programador puede definir excepciones tratadas como objetos y lanzarlas desde algún método de otra clase.

Tenemos la siguiente clase:

```
public class NumerosRojos extends Exception{
 private double cantidad;
 public NumerosRojos(double c){
 this.cantidad = c;
 }
 public double getCantidad(){
 return cantidad;
 }
}
```

Como se puede observar, la clase `NumerosRojos` hereda de la clase `Exception` y nos va a servir para lanzarla desde cualquier otra clase.

Vamos a crear la clase `cuenta`, la cual va a gestionar una cuenta bancaria y tiene los métodos `ingreso` y `retirada` para poder ingresar y sacar dinero de la misma. También tendrá los *getters* `getNumerodecuenta()` y `getSaldo()`.

```
public class Cuenta{
 private double saldo;
 private int numerodecuenta;
 public Cuenta(int num,int sal){
 this.numerodecuenta = num;
 this.saldo = sal;
 }
 public void ingreso(double cantidad){
 saldo = saldo + cantidad;
 }
 public void retirada(double cantidad) throws NumerosRojos{
 if(cantidad > saldo){
 double rojos = cantidad - saldo;
 throw new NumerosRojos(rojos);
 }else{
 saldo = saldo - cantidad;
 }
 }
 public int getNumerodecuenta(){
 return numerodecuenta;
 }
 public double getSaldo(){
 return saldo;
 }
}
```

Se puede observar cómo en el método `retirada` se lanza la excepción `NumerosRojos` si el saldo es menor que la cantidad que se quiere retirar. Observa la diferencia entre `throws` y `throw`. `Throws NumerosRojos` quiere decir que esa función puede lanzar una excepción del tipo `NumerosRojos` mientras que la sentencia `throw NumerosRojos` es la excepción lanzada cuando el saldo es menor a la cantidad demandada.

En la siguiente clase se puede ver cómo se puede probar el código de las dos clases anteriores:

```
public class TestCuenta{
 public static void main(String [] args)
 {
 Cuenta c = new Cuenta(41,100);
 try
 {
 System.out.println("saldo de la cuenta "+c.getNumerodecuenta()+" :"+c.getSaldo());
 c.ingreso(123);
 System.out.println("saldo de la cuenta "+c.getNumerodecuenta()+" :"+c.getSaldo());
 System.out.println("Retiro 85 euros ");
 c.retirada(85);
 System.out.println("saldo de la cuenta "+c.getNumerodecuenta()+" :"+c.getSaldo());
 System.out.println("Retiro 80 euros");
```

```
 c.retirada(80);
 System.out.println("saldo de la cuenta "+c.getNumerodecuenta()+" :"+c.getSaldo());
 System.out.println("Retiro 80 euros");
 c.retirada(80);
 System.out.println("saldo de la cuenta "+c.getNumerodecuenta()+" :"+c.getSaldo());
 }catch(NumerosRojos e){
System.out.println("Error. Falta en la cuenta la siguiente cantidad :"+ e.getCantidad());
 e.printStackTrace();
 }
 }
}
```

El resultado de ejecutar el código anterior se muestra en la siguiente figura:

```
saldo de la cuenta 41 :100.0
saldo de la cuenta 41 :223.0
Retiro 85,50
saldo de la cuenta 41 :138.0
Retiro 80
saldo de la cuenta 41 :58.0
Retiro 80
Error. Falta en la cuenta la siguiente cantidad :22.0
NumerosRojos
        at Cuenta.retirada(Cuenta.java:14)
        at TestCuenta.main(TestCuenta.java:18)
Presione una tecla para continuar . . .
```

Figura 5.9. *Resultado de ejecutar el código de las clases anteriores*

5.4 LIBRERÍAS DE CLASES

Ya en el capítulo 1 se explicó el concepto de librería (paquete) en Java e incluso se comentó cómo se podrían organizar las clases en paquetes a la hora de crear una aplicación.

En una fase temprana de aprendizaje, las librerías no son un concepto importante puesto que primeramente hay que centrarse en cuestiones básicas como la sintaxis, los operadores del lenguaje, etc. Una vez que nos adentramos más en la programación de Java es necesario conocer ciertos aspectos de su API (*Application Program Interface*), que son las librerías compuestas por una serie de clases y algoritmos creados por profesionales, las cuales han sido probadas, depuradas y optimizadas a lo largo de los años y nos pueden ayudar a que nuestra programación sea eficiente y eficaz.

Todo programador debería saber usar la API de Java en la creación de sus aplicaciones.

RECUERDA

No hace falta que conozcas la API a fondo. Todos los programadores acudimos a consultar la documentación por Internet cada vez que tenemos una duda o necesidad.
Puedes buscar información sobre la API de Java tecleando "API Java 8" (o la versión que estés utilizando) en un navegador.

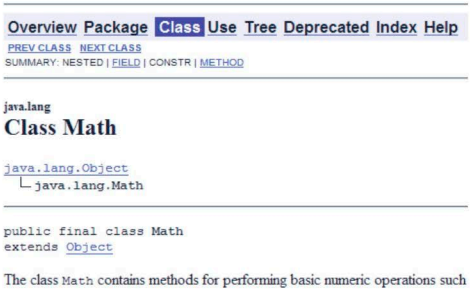


Figura 5.10. *Búsqueda en la API de la clase Math*

Ten en cuenta que hay librerías que no hace falta que las cargues con la sentencia `import`, dado que son parte esencial del lenguaje, como el paquete `java.lang`. Este paquete se importa por defecto y contiene por ejemplo la clase `String` o la clase `System`.

La clase `System` es una de las más utilizadas, sobre todo en los comienzos del aprendizaje de Java, puesto que tiene un campo estático llamado `out` que es el que invocamos cuando ejecutamos el método `println()` en la llamada `System.out.println()`.

Las demás librerías tendrás que declararlas explícitamente en la cabecera de la clase con la sentencia `import` vista en el capítulo 1. Recuerda que cuando importamos `java.awt.*` importamos todas las clases del paquete `java.awt`. Dado que los paquetes pueden llegar a tener cientos de clases, a veces resulta cómodo utilizar el asterisco para no ir enumerando las clases una a una.

En Java, los paquetes y clases se estructuran de manera arborescente y existen dos ramas de paquetes: la rama `java` y la `javax`. La primera es la original de Java mientras que la última es más moderna. El origen de esta diferencia fue que mientras que todas las clases que formaban parte de la API se incluían en el paquete `java`, las que no formaban parte del estándar se iban englobando dentro del paquete `javax` (la *x* era de "extensión"). Con el paso del tiempo, muchas de las clases del paquete `javax` pasaron a formar parte del estándar y al final de este proceso se decidió que la extensión `javax` formara parte del estándar.

Un ejemplo de esto lo veremos en la siguiente imagen, donde aparecen los paquetes `java` y `javax` del API de Java. En estos dos paquetes puedes observar que el subpaquete `sql` aparece en ambos. Eso fue porque hubo clases que al principio formaban parte del estándar y otras formaban parte de las extensiones (en `javax`). Al final de esta anomalía histórica, y por mantener la retrocompatibilidad, existen dos subpaquetes `sql` con diferentes clases en los paquetes `java` y `javax`.

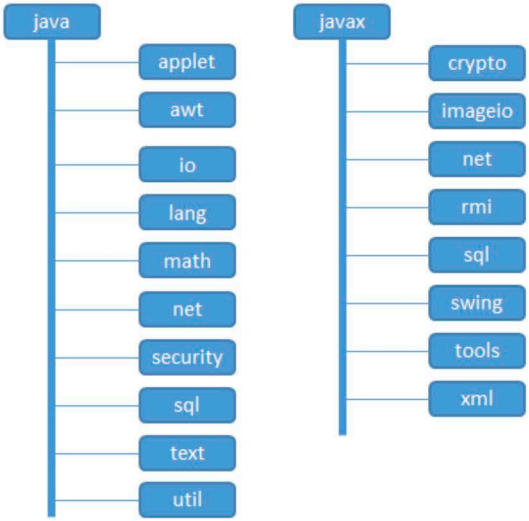


Figura 5.11. *Algunos paquetes de las librerías java y javax*

En la siguiente tabla se explica la funcionalidad de cada una de estas librerías:

Paquete o librería	Descripción
`java.applet`	Librería para desarrollar *applets*.
`java.awt`	Librerías con componentes para el desarrollo de interfaces de usuario.
`java.io`	Librería de entrada/salida. Permite la comunicación del programa con ficheros y periféricos.
`java.lang`	Paquete con clases esenciales de Java. No hace falta ejecutar la sentencia `import` para utilizar sus clases. Librería por defecto

`java.math`	Librería con todo tipo de utilidades matemáticas.
`java.net`	En combinación con la librería `java.io`, va a permitir crear aplicaciones que realicen comunicaciones con la red local e Internet.
`java.security`	Librería que implementa mecanismos de seguridad
`java.sql`	Librería especializada en el manejo y comunicación con bases de datos.
`java.text`	Proporciona clases e interfaces para manejar textos, fechas, números y mensajes.
`java.util`	Librería con clases de utilidad general para el programador.
`javax.crypto`	Librería con múltiples funciones criptográficas.
`javax.net`	Librería que proporciona clases para aplicaciones en red.
`javax.rmi`	Paquete que permite el acceso a objetos situados en otros equipos (objetos remotos).
`javax.sql`	Proporciona la API para el acceso y procesamiento de datos en bases de datos en el lado servidor.
`javax.swing`	Librerías con componentes para el desarrollo de interfaces de usuario. Similar al paquete `awt`.
`javax.tools`	Proporciona las interfaces para herramientas que se puedan invocar desde un programa como un compilador.
`javax.xml`	Define las constantes y funcionalidad de las especificaciones XML.

Tabla 5.1. *Descripción de algunas librerías Java*

5.5 HILOS

Tradicionalmente, la programación se basaba en un único flujo de control o flujo de ejecución en el que se iban ejecutando una serie de órdenes en un único procesador hasta la finalización del programa. Actualmente, con el nuevo hardware y los nuevos sistemas operativos, los hilos de ejecución o multitarea son algo básico.

Aunque el dispositivo donde corra el programa tenga solo un procesador, la programación multihilo va a permitir la existencia de varios flujos de ejecución (tareas que se ejecutan de forma simultánea).

Cada uno de estos hilos pueden realizar la misma tarea (una tarea idéntica) o tareas distintas o complementarias. Veamos esto con un ejemplo de hilo de ejecución: imagina que abres un navegador con varias pestañas y estás lanzando varias consultas o cargando varias páginas web a la vez. Cada una de estas pestañas podría ser manejada por un hilo de ejecución de tal forma que se ejecutan de forma simultánea, pero cada una hace una tarea concreta independiente de la otra.

DIFERENCIA ENTRE MULTITAREA Y MULTIHILO

Multitarea. En un sistema operativo multitarea, varios procesos se pueden estar ejecutando aparentemente al mismo tiempo sin que el usuario lo perciba. La gran mayoría de sistemas operativos actuales son multitarea. Un sistema multitarea permite a la vez estar escuchando música, navegando por Internet y realizando una videoconferencia. El sistema operativo fracciona el tiempo de CPU y lo va repartiendo entre los procesos que lo necesitan de la mejor forma posible. Además de la multitarea aparece el concepto de multihilo.

Multihilo. En ocasiones, un proceso puede tener varios hilos de ejecución de tal manera que un proceso pueda ejecutar varias tareas a la vez. El multihilo se utiliza a veces por eficiencia, debido a que el crear numerosos procesos implica la asignación de gran cantidad de recursos, mientras que muchos hilos pueden compartir los recursos y memoria de un proceso. El multihilo evita además la pérdida de tiempo en cambios de contexto (cambio del sistema operativo para relevar al proceso que se está ejecutando actualmente) al ejecutarse todos los hilos en el mismo contexto.

Por lo tanto, un sistema operativo puede ejecutar varios procesos a la vez y un proceso puede ejecutar varios hilos a la vez.

Los hilos, aunque sean independientes unos de otros, pueden compartir código o datos con otros hilos de ejecución. El compartir datos puede ser beneficioso y peligroso a la vez puesto que hay que tener en cuenta que varios hilos pueden acceder a los mismos datos, con los problemas que ello conlleva.

Los hilos tienen la propiedad de poder pararse, reiniciarse, sincronizarse o bien esperar a otros hilos de ejecución, lo que los convierte en una herramienta bastante potente.

En la programación en Java generalmente hay un hilo principal desde el cual se pueden crear otros hilos independientes. Estos hilos (o *threads*) ejecutan el código que se encuentra dentro de algún método `run()`. Este método `run()` puede estar dentro del propio *thread* o bien dentro de algun objeto. Los objetos pueden implementar el método `run()` dado que existe una interfaz `runnable` que lo hace posible.

RECUERDA

Un *thread* puede ejecutar el método `run()` de otros objetos.

Veamos un ejemplo sencillo de hilos de ejecución. Vamos a crear dos clases, a la primera la llamaremos `hilillo`, hereda de la clase `Thread` y como se puede ver en el siguiente código tiene dos métodos. El primero, o constructor de la clase, sirve para llamar a su clase superior para que almacene el nombre de la clase (parámetro `cad`).

En el segundo método, llamado `run()`, vamos a proceder a sacar por pantalla (salida estándar) el nombre del hilo 5 veces. Para que la ejecución no sea tan rápida puedes observar que se ha introducido un retardo (`Thread.sleep`) de 1000 milisegundos, o, lo que es igual, de un segundo para cada vez que se saque el nombre del hilo por pantalla. El método `run()` también avisa cuando la ejecución del hilo ha terminado.

```
class hilillo extends Thread {
 public hilillo(String cad) {
          super(cad);
 }
 public void run() {
```

```
            for (int i = 0; i < 5; i++) {
                  System.out.println(i + " " + getName());
                  try {
                        Thread.sleep(1000);
                  } catch(InterruptedException ex) {
                        Thread.currentThread().interrupt();
                  }
            }
            System.out.println("Finaliza : " + getName());
      }
}
```

Una vez creada la clase que contiene los hilos de ejecución (el método `run`), habrá que crear un programa que haga uso de esos hilos de ejecución. Para ello creamos la clase `testhilillo`, que, en su método principal, o `main()`, crea hilos de ejecución y los inicializa con el método `start()`.

```
class testhilillo {
 public static void main (String args[]) {
 new hilillo("Juan Carlos Moreno").start();
 new hilillo("Mariamparo Quesada").start();
 new hilillo("Emma Moreno").start();
 }
}
```

El resultado de ejecutar el código anterior se puede ver en la siguiente imagen.

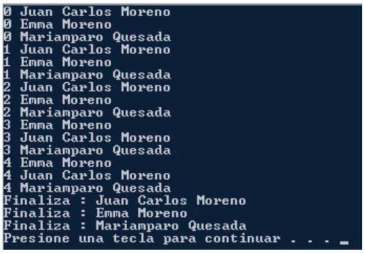


Figura 5.12. *Ejecución del código anterior (hilos)*

Como se puede ver, la ejecución de los hilos es independiente una de otra puesto que el orden no se respeta. Es más, cada vez que ejecutes el programa, el resultado puede ser totalmente diferente.

ACTIVIDADES

- Copia y ejecuta el código de las dos clases anteriores y comprueba cómo el resultado de cada ejecución puede ser diferente.

5.6 PROGRAMACIÓN EN RED

Hasta el momento todos los programas o aplicaciones que hemos ideado se ejecutan en un mismo equipo. No obstante, lo más normal es que muchas de las aplicaciones comerciales se ejecuten en más de un equipo dentro de una red o bien a través de Internet. En estas comunicaciones, generalmente los procesos residen en dos (o más) equipos, uno de ellos es el que inicia las peticiones (generalmente el cliente) y otro se encarga de procesarlas y responderlas (el servidor). Estos roles que hemos visto en el ejemplo anterior forman lo que se denomina arquitectura cliente/servidor.

En la arquitectura cliente/servidor generalmente existe un servidor que se encarga de procesar las peticiones de uno o varios clientes. Esta arquitectura se basa en el protocolo TCP/IP (protocolo dividido en capas) en el cual tanto el servidor como el cliente residen en la capa de aplicación. Ambos interactuan con la capa inferior, llamada "capa de transporte", mediante unos mecanismos que se denominan *sockets*. Estos *sockets* se encuentran definidos en la librería `java.net`.

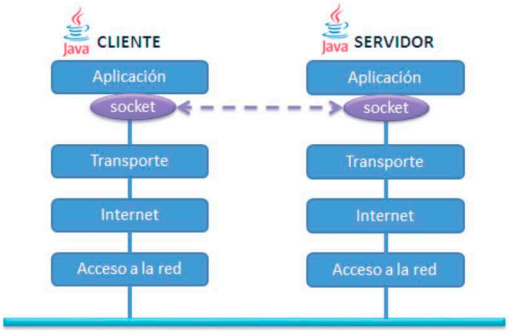


Figura 5.13. *Ejemplo de socket servidor y cliente*

El concepto de programación en red está basado en los *sockets*. Un *socket* es una conexión bidireccional entre dos programas a través de una red. Aunque hay clases diferentes para crear *sockets* del tipo TCP y UDP, generalmente se suelen utilizar los primeros puesto que TCP es un protocolo orientado a conexión con confirmación de recepción de datos y UDP no. Los *sockets* cliente se crearán instanciando la clase `Socket` y los *sockets* servidor se crearán instanciando la clase `ServerSocket`.

La sintaxis de creación tanto del *socket* servidor como del cliente se muestra en el siguiente código:

```
// socket en el lado servidor.
ServerSocket servidor = new ServerSocket (puerto);
// socket en el lado cliente.
Socket socket = new Socket (dirección, puerto);
```

5.7 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Las clases pueden tener superclases y subclases.
b) El polimorfismo se consigue en Java mediante las subclases/superclases y las interfaces.
c) PHP es utilizado como lenguaje base en Wordpress.
d) C++ se considera lenguaje de programación híbrido o multiparadigma.

2 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Java puede simular la herencia múltiple utilizando interfaces.
b) Java es un lenguaje de programación diseñado en los 80 por Bjarne Stroustrup.
c) Objetive-C es un lenguaje orientado a objetos influido por C y SmallTalk.
d) Cuando se programa orientado a objetos se abstraen las características de los objetos que van a formar parte del programa.

3 ¿Cuál de las siguientes afirmaciones es falsa?:
a) En la POO las clases son vistas como una caja negra.
b) PHP es un lenguaje multiparadigma con características de orientación a objetos diseñado por Rasmus Lerdorf.
c) En Java una clase solo puede tener una subclase.
d) El polimorfismo permite crear varias formas del mismo método.

4 ¿Cuál de las siguientes afirmaciones es falsa?:
a) C++ es un lenguaje de programación diseñado en los 80 por Bjarne Stroustrup.
b) En C++ se permite la herencia múltiple, no así en Java.
c) Java puede simular la herencia múltiple utilizando polimorfismo.
d) Python fue diseñado por Guido van Rossum y es un lenguaje multiplataforma como Java.

5 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Java es un lenguaje desarrollado por James Gosling, de IBM, y que apareció en 1995.
b) Según la RAE abstraer es "separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción".
c) Java es un lenguaje desarrollado por James Gosling, de Sun, y que apareció en 1995.
d) VB.NET es la evolución del famoso Visual Basic.

6 Introducción al desarrollo de aplicaciones en el modelo de programación web

Hasta ahora hemos visto aplicaciones en Java. Java es muy versátil y se pueden crear aplicaciones cliente/servidor clásicas o aplicaciones más avanzadas. No obstante, existen otros estándares de programación en la Web como son el HTML, CSS, JavaScript, PHP, etc. En este tema se profundizará en estos conceptos.

6.1 LA ARQUITECTURA MULTICAPA, ESTÁNDARES DE FACTO

Actualmente, a partir de la proliferación de tabletas, *smartphones* y otros dispositivos conectados a Internet, la arquitectura más utilizada es el diseño en **tres niveles** o **tres capas**. En la siguiente figura se puede ver un ejemplo de la misma:

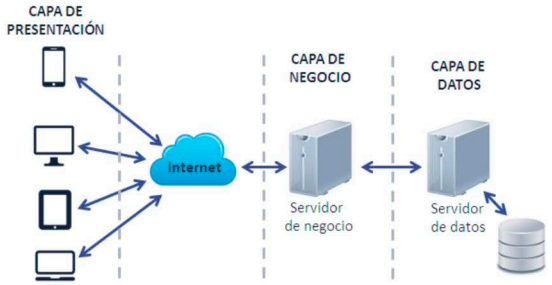


Figura 6.1. *Ejemplo de arquitectura en tres capas*

Generalmente, en la **capa de presentación** o **capa de usuario**, el elemento más importante es la interfaz gráfica, que actualmente suele ser un navegador (o un programa Java). En esta capa no hay demasiado procesamiento, por lo que no se necesita un hardware con mucha potencia como un *smartphone*, *tablet* o similar. Esta capa se comunica a través de la red (local o Internet) con la capa de negocio.

La **capa de negocio** recibe las peticiones de los clientes de la capa de presentación, las procesa y envía las respuestas. En esta capa reside la lógica de negocio, y de ahí su nombre. Esta capa generalmente se comunica con la capa de datos, donde reside el gestor de bases de datos, para leer, escribir y actualizar información.

Por último, en la **capa de datos** reside la información de todo el sistema. Puede estar manejada por uno o varios gestores de bases de datos, generalmente relacionales. Los datos suelen residir en gestores de bases de datos; también se puede encontrar información en ficheros, aunque no es lo normal. Los gestores de bases de datos son la solución más utilizada puesto que operan con un lenguaje estándar como es el SQL y ofrecen un acceso seguro, rápido y eficiente a los datos.

Aunque en la figura anterior cada capa reside en un equipo diferente, hay veces que capa de negocio y capa de datos pueden residir en la misma máquina (incluso podrían residir todas las capas en una misma máquina, aunque no tiene sentido). Dependiendo de lo voluminoso que sea el sistema se dedicarán máquinas específicas para cada función del mismo.

Algunos estándares muy utilizados son un navegador en la capa cliente, un servidor web como Apache o IIS junto con un servidor de aplicaciones (JSP, PHP o ASP) en la capa de negocio, y un servidor de bases de datos relacional como MySQL, Oracle, SQL Server, Informix, etc. Otra posibilidad puede ser utilizar Java en la parte cliente y en la capa de negocio junto con un servidor de bases e datos en la capa de datos. Esta combinación permitiría que se ejecutara también en cualquier plataforma dada la versatilidad de Java.

6.2 LA CAPA DE PRESENTACIÓN: EL LENGUAJE DE HIPERTEXTO

HTML o *Hyper Text Markup Language* es un lenguaje de marcas (*Markup Language*), lo que quiere decir que es un lenguaje que utiliza etiquetas. Estas etiquetas describen el contenido que engloban. Si has tenido la ocasión de abrir un fichero HTML, lo que vas a encontrar es una serie de etiquetas y texto plano. Los ficheros HTML también se denominan "páginas web" y tienen extensión `html` o `htm`.

Las etiquetas en HTML son palabras clave, las cuales están delimitadas por los símbolos `<y>`. Generalmente las vas a encontrar pareadas, por ejemplo `<strong>` y `</strong>`. Como puedes observar, la primera etiqueta no tiene el símbolo / pero la segunda sí. La primera se llama "de comienzo" y la segunda "de cierre". Un ejemplo de etiqueta sería el siguiente:

```
<strong>editorial RA-MA</strong>
```

En el ejemplo anterior podemos distinguir entre etiquetas HTML y elementos HTML. Un elemento HTML es aquella información que se encuentra entre la etiqueta de comienzo y la de cierre. En el caso anterior, el texto "editorial RA-MA".

LOS NAVEGADORES (*WEB BROWSER*)

Los navegadores tienen como misión leer las páginas web y mostrar el contenido de acuerdo a las órdenes contenidas en el HTML. Obviamente, aunque las páginas web tienen etiquetas, estas nunca las mostrará el navegador. Las etiquetas ayudarán al navegador a saber cómo mostrar dicha información.

Figura 6.2. *Navegador Google Chrome*

En el año 2012 salió al mercado el último estándar de HTML: el **HTML5**. HTML5 reemplaza a las versiones anteriores de HTML y viene cargado de nuevas funcionalidades (animación, vídeos, gráficos, nuevas etiquetas de contenido, nuevos elementos en formularios, etc.) sin que se necesite instalar *plugins* adicionales (como Flash). Al ser un lenguaje multiplataforma va a funcionar en un *smartphone*, *tablet*, PC, televisión, etc.

A continuación se muestra un documento con las mínimas etiquetas necesarias:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>T&iacutetulo</title>
</head>
<body>
Contenido del documento
</body>
</html>
```

Estructuralmente, la página anterior sería algo parecido a la siguiente figura:

Figura 6.3. *Estructura de un documento HTML*

Veamos más detenidamente las etiquetas del código anterior:

- **`<!DOCTYPE html>`**. Esta etiqueta es una declaración de HTML5 e indica al navegador el tipo de HTML utilizado y la versión.
- **`<html>`**. Esta etiqueta es el contenedor principal de la página HTML.
- **`<head>`**. Esta etiqueta delimita la cabecera de la página web. En la cabecera podemos incluir el título, el estilo, *scripts*, metadatos, etc.

- **<meta charset="UTF-8">**. Meta es una etiqueta que indica que se van a incluir metadatos sobre el documento HTML. En este caso se especifica que la codificación de los caracteres de la página se hará con UTF-8.
- **<title>**. Esta etiqueta delimita el título de la página web.
- **<body>**. Esta etiqueta delimita el contenido de la página web.

MAYÚSCULAS O MINÚSCULAS EN LAS ETIQUETAS

Las etiquetas no entienden de mayúsculas y minúsculas, para HTML `<body>` es igual que `<BODY>`, pero el W3C (World Wide Web Consortium) recomienda utilizar minúsculas.

6.2.1 LAS ETIQUETAS MÁS IMPORTANTES EN HTML

En HTML hay cientos de etiquetas. En este apartado hemos resumido las más importantes que tienes que conocer para la programación web y para crear páginas HTML.

6.2.1.1 Párrafos y saltos de línea

Para formatear texto en párrafos tenemos las etiquetas `<p>` y `</p>`, las cuales nos van a ayudar a diferenciar un párrafo de otro en el contenido de nuestras páginas web. Si queremos realizar un salto de línea basta con escribir la etiqueta `<br>`. Ten en cuenta que aunque escribas saltos de línea en el texto de tu página web, estos no van a aparecer cuando la abras con el navegador salvo que los hagas explícitos con la etiqueta `<br>`. Como ya se ha explicado, el navegador lee el código HTML de la página web, lo interpreta y lo visualiza.

```
<p> This is a paragraph </p>
<br>
```

LOS ATRIBUTOS DE LAS ETIQUETAS

Muchas etiquetas, como veremos más adelante, tienen atributos y nos van a servir de gran ayuda para especificar el formato concreto o bien para completar la información necesaria. Por ejemplo, si queremos referenciar una imagen, tendremos que indicar dónde está alojada, y si queremos añadir un enlace, deberemos indicar la dirección. También los atributos nos van a servir para especificar el estilo, para ello se utilizarán atributos como `class` (para especificar la clase) o `style` (para especificar un estilo *inline* CSS).

6.2.1.2 Formato del texto

Una vez que sabemos cómo dividir la información en párrafos necesitaremos conocer otras etiquetas para poner parte del texto en negrita, cursiva, subrayado, etc. A continuación citaremos las etiquetas más usuales para especificar el formato del texto:

- **<b>** Para expresar negrita.
- **** Para definir un texto resaltado.
- **<i>** Para expresar cursiva.
- **<small>** Para hacer que el texto aparezca en pequeño.
- **** Para definir un texto resaltado.
- **<sub>** Para poner el texto en subíndice.
- **<sup>** Para poner el texto en superíndice.
- **<mark>** para marcar un texto

Un ejemplo de lo visto hasta ahora sería el siguiente:

```
<p>En un lugar de la <b>Mancha</b> de cuyo <i>nombre</i> no quiero <strong>acordarme,</
strong></p>
<p>no ha mucho <small>tiempo</small> que viv&iacute;a un <mark>hidalgo</mark> de los de
lanza en <em>astillero</em>,
adarga antigua, roc&iacute;n <sup>flaco</sup> y galgo <sub>corredor</sub>. </p>
```

El ejemplo anterior visto en un navegador sería el siguiente:

En un lugar de la **Mancha** de cuyo *nombre* no quiero **acordarme,**

no ha mucho tiempo que vivía un hidalgo de los de lanza en *astillero*, adarga antigua, rocín flaco y galgo corredor.

Figura 6.4. *Detalle de la vista del ejemplo anterior en un navegador*

LOS CARACTERES ACENTUADOS EN HTML

Como podrás ver en el ejemplo anterior, las palabras acentuadas como "vivía" y "rocín" tienen los siguientes códigos: `&iacute;`
Dicho código le sirve al navegador para interpretar que tiene que sustituirlo por un carácter acentuado. En este caso, la *í*. Si fuera una *á*, utilizaríamos el código `&aacute;` y así con las demás vocales.

6.2.1.3 Encabezados

En HTML podemos crear encabezados de mayor a menor importancia con las etiquetas `<h1>` hasta `<h6>`.

Un ejemplo de encabezados en una página web sería el siguiente:

```
<h1>Encabezado principal</h1>
<h2>Encabezado secundario</h2>
<h3>Encabezado tercero</h3>
```

En un navegador, el resultado sería el siguiente:

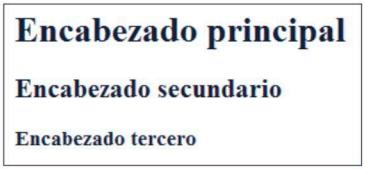


Figura 6.5. *Detalle del ejemplo anterior en un navegador*

6.2.1.4 Enlaces e imágenes

En las páginas web es muy frecuente insertar enlaces a otras páginas web e imágenes (de hecho, los enlaces son la esencia del hipertexto).

Veamos un ejemplo de un enlace dentro de una página HTML

```
<a href="http://myfpschool.com">Enlace a Myfpschool</a>
```

Como se puede observar, en el atributo `href` se incluye la página web que se desea abrir, y entre las etiquetas `<a>` y `</a>` el texto que aparecerá en el navegador.

En caso de que queramos incorporar una imagen a nuestra página web, incluiremos el siguiente código:

<img src="http://myfpschool.com/wp-content/uploads/2013/11/librobricolajeinformatico.gif" alt="Bricolaje del hardware" width="100" height="139">

Donde el atributo `src` contiene la dirección de la imagen que se va a visualizar, `alt` es el texto alternativo que se mostrará, y `width` y `height` la anchura y altura respectivamente.

Veamos un ejemplo con imágenes y enlaces:

```
<a href="http://myfpschool.com">Enlace a Myfpschool</a>
<br><br>
<img src="http://myfpschool.com/wp-content/uploads/2013/11/librobricolajeinformatico.
gif" alt="Bricolaje del hardware" width="100" height="139">
```

En un navegador, el resultado sería el siguiente:

Figura 6.6. *Detalle del ejemplo anterior en un navegador*

BUENA PRÁCTICA

Siempre que puedas, fija la anchura y altura (`width` y `height`) de las imágenes en tu página web. Sin esas dimensiones, el navegador no conoce el tamaño de la imagen y el aspecto que ofrecerá una vez cargada la página puede que no sea el que tú quieres. Además, mientras carga la página, muchas veces el navegador reorganiza los elementos de la misma si no introduces el tamaño de las imágenes, ofreciendo así un efecto poco profesional.

6.2.2 NOVEDADES EN HTML5

Como ya hemos explicado, HTML5 define nuevos elementos dentro del lenguaje, algunos de ellos sirven por ejemplo para definir las partes de una página web (esta capacidad mejora el estándar existente, puesto que antes el programador tenía que jugar con los atributos `id` y `class` y ahora ya se definen claramente). A continuación se muestra un esquema de estos nuevos elementos.

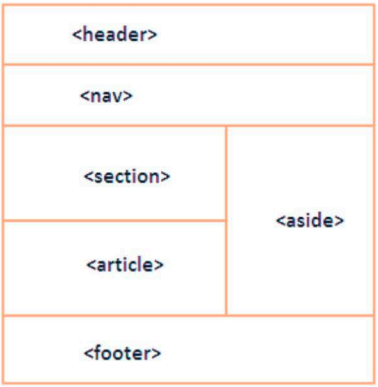


Figura 6.7. *Estructura de los nuevos elementos HTML5*

Otra característica que ha mejorado bastante en HTML5 son los formularios. En el estándar anterior había que validar ciertos campos con JavaScript o utilizar ciertas librerías o código para realizar cosas que ahora son ofrecidas de forma estándar.

Veamos un ejemplo de nuevos elementos que se pueden incluir en formularios:

```
<form action="formulario.php" method="get">
Calificaci&oacute;n: 0<input type="range" name="calif" min="1" max="10">10
<br>
Fecha: <input type="date" name="fecha">
<br>
Hora: <input type="time" name="hora">
<br>
Buscar: <input type="search" name="buscar"><br><br>
<input type="submit">
</form>
```

Como se puede ver, el significado de los campos anteriores es fácil de adivinar. A continuación se muestra el aspecto del formulario anterior visto en un navegador:

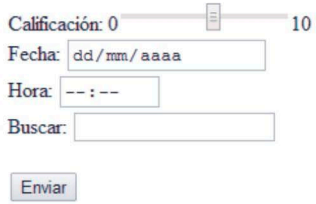


Figura 6.8. *Nuevos elementos en formularios*

Otra de las nuevas características en HTML5 son los gráficos. El nuevo estándar ofrece un nuevo y potente elemento como son los <canvas> o lienzos.

Veamos un ejemplo de lienzo con HTML5:

```
<canvas id="lienzo" width="275" height="100" style="border:5px solid #d3d3d3;">
Lo sentimos pero tu navegador no soporta lienzos en HTML5.</canvas>
<script>
var c=document.getElementById("lienzo");
var cont=c.getContext("2d");
// Creando el gradiente
var grd = cont.createRadialGradient(120,60,25,120,60,100);
grd.addColorStop(0,"white");
grd.addColorStop(1,"lightgray");
// Rellenando el gradiente
cont.fillStyle = grd;
cont.fillRect(0,0,275,100);
// Rellenando con texto el lienzo
cont.font="30px Courier";
cont.strokeText("Myfpschool.com",10,60);
</script>
```

En el código anterior se puede observar que se utiliza la etiqueta `canvas` para crear el lienzo con una anchura y altura determinadas (`width` y `height`) a la cual le hemos dotado de un borde gris de 5 píxeles de grosor (`style="border:5px solid #d3d3d3;`). Como podrás observar, para crear el gradiente y rellenar el lienzo con el texto se utiliza JavaScript (`<script></script>`). JavaScript es un lenguaje de *scripting* que podemos utilizar en las páginas web y lo estudiaremos en el siguiente apartado.

Figura 6.9. *Ejemplo anterior visto en un navegador*

En el caso del vídeo, hasta HTML5 no había un estándar para vídeos en páginas web. Hasta ese momento había que utilizar *plugins* como Flash con el peligro de seguridad que eso conllevaba (de hecho, iOS no utilizaba Flash por este motivo). Con HTML5 aparece la etiqueta `<video>`, la cual está soportada prácticamente por la totalidad de navegadores. Además, el control permite pausar, manejar el volumen, reanudar, etc.

En el siguiente ejemplo se muestra el código que habría que insertar en la página web para introducir un vídeo:

```
<video width="420" height="340" controls>
 <source src="peli.mp4" type="video/mp4">
 <source src="peli.ogg" type="video/ogg">
Tu navegador no soporta la etiqueta v&iacute;deo.
</video>
```

Como puedes ver, en la etiqueta `video` se incluyen los atributos de altura y anchura para que la página web reserve tamaño para el vídeo. Si no incluyes nada, el navegador no conoce las dimensiones del vídeo y es probable que tu página web luego no quede bien en el navegador.

Con el audio pasa lo mismo que con el vídeo en HTML5. El control de audio es muy parecido al de vídeo, es posible pausar, reanudar, ajustar volumen, etc.

En el siguiente ejemplo se muestra el código que habría que insertar en la página web para introducir un audio y el aspecto del control en la página:

```
<audio controls>
 <source src="sonido.ogg" type="audio/ogg">
 <source src=" sonido.mp3" type="audio/mpeg">
Tu navegador no soporta la etiqueta audio.
</audio>
```

Figura 6.10. *Control de audio en HTML5*

6.3 LA CAPA DE PRESENTACIÓN AVANZADA: JAVASCRIPT Y CSS

Actualmente no se entiende una página web sin CSS para adaptar la apariencia o JavaScript para hacer más dinámico el contenido. De hecho, HTML5, CSS3 y JavaScript son los tres pilares básicos que debería dominar cualquier programador de páginas web; además de otros CMS o gestores de contenido, como Wordpress por ejemplo.

En los siguientes apartados se dará una introducción a CSS y JavaScript. Obviamente, ambos lenguajes son mucho más complejos que lo que vamos a ver en este libro. Ten en cuenta que una vez que adquieras los conceptos básicos de ambos lenguajes puedes encontrar muchos manuales y muy buenos por Internet que completarán tus conocimientos de CSS y JavaScript.

6.3.1 CSS

CSS (*Cascading Style Sheets*) nació por la necesidad de solucionar un problema existente con los estándares anteriores de HTML debido a que había que replicar los atributos de muchas etiquetas en todo el código. Los desarrolladores de sitios grandes vivían verdaderos problemas cuando tenían que especificar atributos como la fuente o el color de la misma para cada página web. Las páginas se hacían muy grandes y poco operativas debido a que había que multiplicar mucho código dentro de la página.

Con CSS, los ficheros HTML quedan más limpios y es posible guardar el código CSS en ficheros externos CSS.

Generalmente, con CSS se trabaja definiendo los estilos en ficheros separados (con extensión `.css`), los cuales hacen cambiar la apariencia y colocación de los elementos de la página web. Cuando se quiere modificar algún contenido de forma general, se editan estos ficheros CSS, se realizan los cambios y no hay que modificar las páginas web una a una.

FORMAS DE INSERTAR CSS EN TUS PÁGINAS HTML

Hay tres formas de insertar CSS en las páginas HTML:

- **Con ficheros externos CSS**. Es la opción recomendada. Los estilos se almacenan en ficheros separados y al modificarlos surten efecto en las páginas web que tengan como estilo dichos ficheros.
- **Dentro de la página web**. Poco recomendable. Es el formato del ejemplo siguiente. El estilo está dentro de la propia página web entre las etiquetas `<style>`. Es una opción válida cuando tenemos una única página web (un solo fichero) pero imposible de mantener con muchas.
- **Código *inline* o estilo *inline***. Totalmente desaconsejado. Se utiliza la propiedad `style` de las etiquetas. Se puede emplear para algo puntual, pero cuando la página es muy grande o hay muchos ficheros HTML, el uso de este tipo de CSS hace el código bastante cargado y caótico.

Veamos un ejemplo de página web con CSS contenido en la etiqueta <style>:

```
<!DOCTYPE html>
<html>
<head>
<style>
h1 { color:gray; text-align:center; }
p { font-family:"Arial"; text-align:center; font-size:18px; }
</style>
</head>
<body>
<h1>myfpschool.com</h1>
<p>Tu sitio de inform&aacute;tica y tecnolog&iacute;a.</p></body>
</html>
```

Como puedes observar en el código anterior, el código CSS va incluido en la etiqueta <style> situada en la cabecera (<head>) de la página web.

Figura 6.11. *Aspecto final del código anterior*

6.3.1.1 Sintaxis de las reglas CSS

El código CSS está dividido en reglas. Cada una de las reglas tiene dos partes diferenciadas:

- El **selector**. Define el elemento HTML del cual queremos definir el estilo.
- El **bloque de declaración**. El bloque de declaración contiene una o varias declaraciones separadas unas de otras por punto y coma ";". Cada declaración —como se puede ver en la figura siguiente— está compuesta por la propiedad y el valor separados por dos puntos ":".

En la siguiente figura se ve gráficamente una regla del ejemplo anterior:

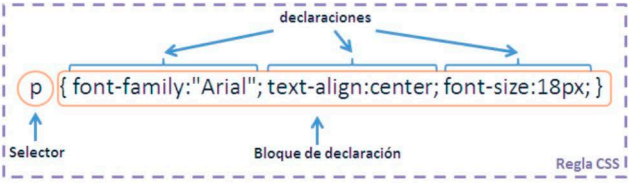


Figura 6.12. *Esquema de la sintaxis de las reglas CSS*

6.3.1.2 Los selectores

Los selectores sirven en las reglas CSS para identificar o buscar los elementos HTML, los cuales se pueden buscar por su propio tipo, identificador (`ID`), clase (`class`), atributo, valores del atributo, etc.

Un ejemplo de selector utilizando un identificador sería el siguiente:

```
<!DOCTYPE html>
<html>
<head>
<style>
h1 { color:gray; text-align:center; }
#parra { font-family:"Arial"; text-align:center; font-size:18px; }
</style>
</head>
<body>
<h1>myfpschool.com</h1>
<p id="parra">Tu sitio de inform&aacute;tica y tecnolog&iacute;a.</p>
<p>P&aacute;rrafo normal.</p>
</body>
</html>
```

Como se puede ver en el ejemplo anterior se crea un identificador `parra` en la sección de estilo de la página web y luego se lo asignamos como identificador al párrafo primero. El resultado del ejemplo anterior sería el siguiente:

Figura 6.13. *Ejemplo del código anterior visto en un navegador*

En el ejemplo anterior se crean dos párrafos, uno afectado por el identificador y otro no.

Otra posibilidad es utilizar el selector `class` para especificar una clase concreta. Para ello en HTML tenemos que utilizar el atributo `class`.

Para definir una clase se coloca el nombre específico de la clase precedido de un punto. Un ejemplo de regla sería la siguiente:

```
.centrado{ font-family:"Arial"; text-align:center; font-size:18px; }
```

Podemos hacer que todos los elementos cuya clase sea "`centrado`" hereden los atributos de dicho selector:

```
<h1 class="centrado">myfpschool.com</h1>
<p class="centrado">Tu sitio de inform&aacute;tica y tecnolog&iacute;a.</p>
```

Imaginemos que solamente queremos que hereden los elementos HTML de la clase `centrado` pero que sean del tipo `párrafo`. Entonces en la sección `style` escribiríamos la siguiente regla:

```
p.centrado{ font-family:"Arial"; text-align:center; font-size:18px; }
```

Veamos gráficamente cuál sería el resultado de los ejemplos anteriores, el primero con el selector "`.centrado`" y el segundo con el selector "`p.centrado`".

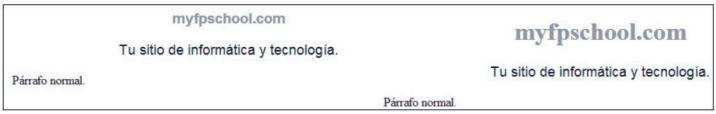


Figura 6.14. *Diferencias entre los dos ejemplos anteriores*

6.3.1.3 CSS3

Los nuevos navegadores ya soportan CSS3, el cual tiene nuevas propiedades que se han dividido en módulos. Algunos de los nuevos módulos que se han añadido son los siguientes:

- Efectos de texto.
- Transformaciones 2D y 3D.
- Animaciones.
- Selectores.
- Modelos de cajas.

Veamos algunos ejemplos de las nuevas especificaciones de CSS3.

Border-radius

Con `border radius` podremos crear cuadros redondeados y con efectos. Muchas de estas opciones se hacían antiguamente con imágenes y había que utilizar Gimp, Photoshop u otro programa de tratamiento de imágenes. Actualmente, con CSS3 ya no hace falta. Veamos un ejemplo de esto:

```
<style>
.nota_comando {
    font-family: Courier;
    color: white;
    border: 2px solid black;
    padding: 10px 40px;
    background: black;
    width: 80%;
    border-radius: 5px;
    margin-bottom: 15px;
    margin-top: 15px;
}</style>
….
<div class="nota_comando">myfpschool.com</div>
```

El resultado de incluir el código anterior en nuestra página web sería el siguiente:

Figura 6.15. *Resultado del ejemplo anterior utilizando border-radius*

Veamos ahora otro efecto parecido, el cual crea un cuadro sombreado.

Box-shadow

Un ejemplo de `box-shadow` sería el siguiente:

```
div{
    width:300px;
    height:100px;
    background-color:lightgray;
    box-shadow: 10px 10px 5px #888888;
    padding: 10px 40px;
}
….
<div><h1>myfpschool.com</h1></div>
```

El resultado de incluir el código anterior en nuestra página web sería el que se muestra a continuación:

Figura 6.16. *Resultado del ejemplo anterior utilizando box-shadow*

Como puedes ver en los ejemplos, además de las propiedades `border-radius` y `box-shadow` hemos utilizado otras propiedades para especificar la anchura, altura, color de fondo, márgenes, etc.

6.3.2 JAVASCRIPT

JavaScript es uno de los lenguajes imprescindibles cuando se desarrollan páginas web. Hasta ahora hemos visto que con HTML definimos el contenido y los elementos de una página web, con CSS podíamos adornar el contenido con un estilo previamente definido, y, por último, con JavaScript dichas páginas pasarán de ser algo estático a algo dinámico.

A continuación veremos el código de la típica página que muestra "hola mundo" cuando se pulsa un botón:

```
<!DOCTYPE html>
<html>
<body>
<h1>T&iacute;pico ejemplo hola mundo</h1>
<button type="button" onclick="Saluda()">Dale y te saludo</button>
<p id="saluda"></p>
<script>
function Saluda() {
 document.getElementById("saluda").innerHTML = "Hola mundo";
}
</script>
</body>
</html>
```

Veamos paso a paso cómo funcionan algunas de las órdenes de esta página web:

```
<button type="button" onclick="Saluda()">Dale y te saludo</button>
```

En la anterior línea de código se puede observar cómo se crea un botón cuyo evento `onclick` (cuando es pulsado) hace que se ejecute la función `Saluda()`. Dicha función deberá ser definida posteriormente en un *script* en JavaScript.

```
<p id="saluda"></p>
```

En la anterior línea de código se puede observar cómo se ha definido con identificador "saluda" un elemento párrafo. Este elemento será accedido mediante JavaScript y se modificará en el *script* posterior.

```
<script>
function Saluda() {
 document.getElementById("saluda").innerHTML = "Hola mundo";
}
</script>
```

Como se puede observar, el código anterior es un *script* (obviamente, porque va delimitado con las etiquetas `<script>`) en JavaScript y realiza una función muy sencilla: acceder dentro del DOM con el método `getElementById` (este concepto se explicará a continuación) y modificar el elemento "saluda" asignándole el literal "Hola mundo":

EJERCICIO PROPUESTO

Como ejercicio propuesto el lector tendrá que copiar el código anterior, ejecutarlo y ver cómo funciona.

El resultado de incluir el código anterior en nuestra página web sería el que se muestra a continuación:

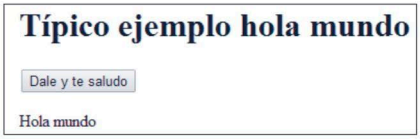


Figura 6.17. *Vista en el navegador del ejemplo anterior*

EL DOM

Cuando el navegador carga una página web, crea un DOM; es decir, un modelo de la página, el cual puede ser representado por un árbol. Ejemplo de un DOM sencillo podría ser el siguiente:

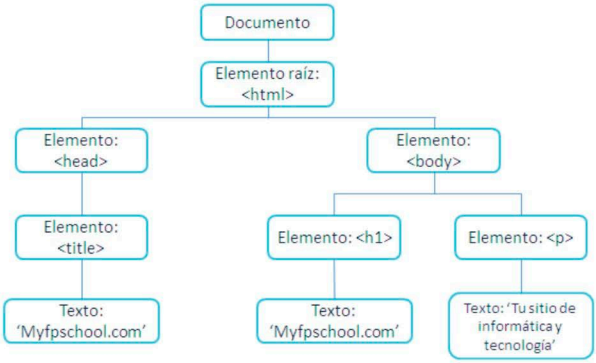


Figura 6.18. *Ejemplo de DOM*

JavaScript puede acceder a todos los objetos del DOM y los puede modificar, añadir, eliminar, etc. Algunas de las acciones que puede realizar JavaScript son:

- Cambiar cualquier elemento HTML en la página.
- Cambiar los atributos de los elementos HTML de la página.
- Cambiar el estilo CSS de la página.
- Eliminar cualquier elemento HTML y sus atributos.
- Añadir nuevos elementos HTML y atributos.
- Reaccionar ante cualquier evento que se produzca en la página.
- Crear nuevos eventos HTML.

6.3.2.1 Accediendo al DOM

En el apartado anterior hemos visto el concepto de DOM y cómo acceder al mismo mediante JavaScript. A continuación veremos otro ejemplo en el que se puede ver cómo además de acceder a un elemento se puede modificar cualquiera de sus atributos.

```
<!DOCTYPE html>
<html>
<body>
<h1>Modificando un elemento del DOM</h1>
<p id="demo" onclick="grande()">Pulsa sobre mí a ver que pasa.</p>
<script>
function grande() {
 var x = document.getElementById("demo");
 x.innerHTML = "¡Vaya he crecido!";
 x.style.fontSize = "35px";
 x.style.color = "gray";
}
</script>
</body>
</html>
```

Se puede ver en el ejemplo anterior cómo se pueden modificar en un *script* los atributos de fuente, color y texto de un elemento HTML. Además se le añade interactividad al texto porque al pulsar sobre él se ejecutará la función `grande`, que hace que el texto anterior cambie de tamaño, color y contenido.

El resultado de ejecutar la página web anterior es el siguiente:

Figura 6.19. *Vista en el navegador del ejemplo anterior*

En la figura anterior se puede ver el antes (izquierda) y el después (derecha) tras pulsar sobre el texto.

6.3.2.2 Conceptos que deben tenerse en cuenta con JavaScript

En este apartado veremos algunos conceptos a tener en cuenta cuando se programa con JavaScript:

La sintaxis

La sintaxis como has podido observar es muy parecida a la de Java por no decir idéntica salvo algunas diferencias. Como ya has aprendido en capítulos anteriores la sintaxis de Java, cualquier código en JavaScript te parecerá familiar.

Operadores, literales, sentencias, etc, funcionan igual que en Java. La mayoría de las palabras clave son las mismas. Dado que JavaScript es un lenguaje de *scripting*, las variables se tratan de otra manera.

Las variables

Dado que JavaScript es un lenguaje de *scripting* y los *scripts* deben ser interpretados por los navegadores, las variables funcionan de forma algo diferente a como lo hacen en Java.

No hace falta definir las variables, por ejemplo en un *script* puedo hacer:

```
x = document.getElementById("demo");
```

y también

```
var x = document.getElementById("demo");
```

y funcionará de igual manera. No obstante, la segunda línea de código es más correcta que la primera puesto que se define una variable y al programador le resultará mucho más sencillo seguir y comprender el código. Además, una programación correcta siempre evita errores inesperados.

Localización de los scripts

Como hemos visto, los *scripts* en JavaScript se situan entre las etiquetas `<script>` y en el caso anterior lo hemos colocado dentro del elemento `<body>`. También es posible colocarlos dentro del elemento `<head>` y funcionarán correctamente (o en ambos, en `<head>` y en `<body>`).

Además, se pueden escribir tantos *scripts* como necesite nuestra página. Eso sí, es mejor separar JavaScript del código HTML para que la página sea más legible y fácil de mantener en un futuro.

CONSEJO

Coloca los *scripts* al final de la sección `<body>` de la página web. De esa manera se reducirá el tiempo de visualización y la página se cargará más rápido en el navegador.

Escribir en el documento y en la consola

Hemos visto en apartados anteriores cómo acceder al DOM y añadir texto a elementos ya existentes. A continuación se mostrará un ejemplo de cómo JavaScript puede escribir texto en una página web sin tener que acceder a un elemento previamente creado:

```
<script>
document.write("<h3>myfpschool.com</h3>");
</script>
```

En el anterior código se puede ver cómo con JavaScript puedes crear elementos (en este caso un elemento `<h3>`) y visualizarlos en la página web.

También en JavaScript se puede escribir en la consola. La consola no se visualiza en el navegador pero es útil para cuando se quieren mostrar mensajes para el programador o para el técnico.

Un ejemplo de mensaje enviado a la consola sería el siguiente:

```
<script>
console.log("página terminada");
</script>
```

En el código anterior se puede observar cómo el programador envía a la consola el mensaje `"página terminada"`. Para ver este mensaje es imprescincible acceder a la consola. En los navegadores más frecuentes (Chrome, IE o Firefox) se accede pulsando **F12** y seleccionando dentro del menú de **Debug** la opción **Console**.

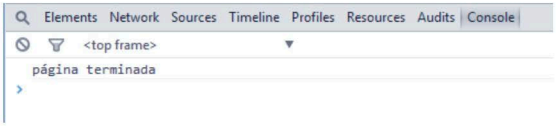


Figura 6.20. *Aspecto de la consola una vez cargada la página web del ejemplo anterior*

Los tipos de datos en JavaScript

Como hemos visto en los ejemplos anteriores, JavaScript puede manejar muchos tipos de datos (números, *strings*, *arrays*, objetos, etc.). Además, las variables son interpretadas y el tipo de datos se lo asigna el intérprete en el momento de ejecución.

Veamos ejemplos de tipos de datos en JavaScript:

```
var edad = 9; // edad es ahora de tipo Number
var saldo = 45e5; // saldo es numérico con valor 45000
var nombre = "Emma"; // nombre es de tipo String
var nombres = ["Emma", "Amparo", "Chelo"]; //nombres es un Array con tres valores
var alumno = {nombre:"Pedro", numMatric:651}; // alumno es un Objecto
alumno = null; //vaciamos el contenido del objeto alumno
```

RECUERDA

Cuando creamos una variable de la siguiente forma (con **new**):

```
var edad = new Number;
```

Creamos un objeto de tipo `Number` al que apunta la variable `edad`.

6.3.2.3 Eventos en JavaScript

Ya hemos visto en ejemplos anteriores cómo JavaScript podía reaccionar a eventos que les ocurrían a los elementos HTML.

LOS EVENTOS

Un evento es algo que ocurre en una página web. Algunos eventos pueden ser el cambio de un campo de texto de la página, que la página se ha terminado de cargar o que se ha hecho clic sobre un elemento de la página, como, por ejemplo, un botón.

Como hemos visto, la codificación de los eventos en un elemento HTML sigue el siguiente patrón:

< *elemento_HTML evento* = '*código JavaScript*' >

Figura 6.21. *Patrón de codificación de eventos en JavaScript*

La forma más elegante de codificar programas en JavaScript es utilizar funciones fuera de la página web en un fichero separado o bien al final del `body` para que la página cargue antes. No obstante, se puede insertar código dentro de la propio atributo de evento. Un ejemplo de esto sería el siguiente:

```
<button onclick="x='hola mundo'; getElementById('saludo').innerHTML=x">Pulsa aquí</
button>
<p id="saludo"></p>
```

Como puedes ver, es posible incluir código dentro del mismo evento `onclick` en el elemento HTML. No obstante, es más recomendable utilizar funciones porque el código está más localizado y es más limpio.

Algunos eventos que pueden ocurrir a los elementos HTML son los siguientes:

- **`onload`**. El navegador termina de cargar la página.
- **`onclick`**. El usuario hace clic al `element`.
- **`onchange`**. Se produce un cambio en un elemento HTML.
- **`onmouseover`**. El usuario mueve el ratón sobre el `element`.
- **`onmouseout`**. El usuario estaba moviendo el ratón sobre el `element` y deja de hacerlo, moviéndolo a otro lado.
- **`onkeydown`**. El usuario pulsa una tecla.

6.3.2.4 Objetos en JavaScript

Los objetos en JavaScript siguen la filosofía de Java con algunas pequeñas diferencias. De hecho, prácticamente todo en JavaScript son objetos (*strings*, *arrays*, etc.). Como veíamos en capítulos anteriores, los objetos son unas estructuras a las que se les han añadido propiedades y métodos, entendiendo por propiedades los datos del objeto; los métodos serán acciones que puedan ejecutar dichos objetos.

Un ejemplo de objeto en JavaScript sería el siguiente:

```
<p id="alumn"></p>
<script>
var alumno = {
 nombre: "María Amparo",
 apellidos: "Quesada Heredia",
 nummat: 4321,
 imprimir: function (){return this.nummat+ ": " +this.nombre+ " " + this.apellidos}
};
document.getElementById("alumn").innerHTML = alumno.imprimir();
</script>
```

Comentemos paso a paso el código anterior:

```
<p id="alumn"></p>
```

En la línea anterior creamos un elemento `<p>` (párrafo) al que le asignamos el identificador `"alumn"`. Este identificador nos va a servir más adelante para acceder a dicho elemento dentro del DOM.

```
<script>
var alumno = {
 nombre: "María Amparo",
 apellidos: "Quesada Heredia",
 nummat: 4321,
 imprimir: function (){return this.nummat+ ": " +this.nombre+ " " + this.apellidos}
};
```

En el código anterior creamos el objeto `alumno`. Como se puede ver, el objeto `alumno` tiene tres atributos (`nombre`, `apellidos` y `nummat`). También tiene asociado un método llamado `imprimir` que muesta por pantalla toda la información de dicho objeto. Ese método es invocado en la línea de código siguiente:

```
document.getElementById("alumn").innerHTML = alumno.imprimir();
```

En la línea de código anterior se puede ver cómo el *script* accede al DOM, modifica el valor del elemento `"alumn"` y le asigna el contenido del objeto `alumno`. El resultado de ejecutar la página web en el navegador será mostrar el siguiente texto:

4321: María Amparo Quesada Heredia

6.4 LENGUAJES DE PROGRAMACIÓN WEB (JSP, SERVLETS, ASP Y PHP)

En el apartado anterior hemos visto los lenguajes más utilizados desde la parte cliente (HTML, CSS, JavaScript). Todo el código escrito en esos lenguajes anteriores se ejecuta en el navegador del cliente. También existen lenguajes de programación que se ejecutan dentro del servidor, como son los *servlets*, JSP, ASP y PHP.

De todos estos lenguajes anteriores, PHP (*PHP Hypertext Preprocessor*) es el más utilizado puesto que con este lenguaje se han escrito muchos sistemas ampliamente utilizados, como Wordpress, Joomla, etc. También Facebook está escrito en este lenguaje, lo que muestra lo potente y eficiente que puede llegar a ser.

Con PHP podrás recoger datos de formularios, enviar y recibir *cookies*, acceder a bases de datos, encriptar datos, generar contenido dinámico, trabajar con ficheros en el lado del servidor, etc.

PHP en última instancia envía una salida HTML al servidor web a la vez que imágenes, ficheros PDF, Flash, XML o XHTML.

Veamos a continuación algunos de los fundamentos de este lenguaje.

Figura 6.22. *Logo de XAMPP*

ANTES DE COMENZAR

Para empezar a programar y estudiar PHP necesitarás tener instalado en un equipo un servidor PHP. Te aconsejamos instalar XAMPP (las siglas vienen de X – Linux, A – Apache, M – MySQL, P – PHP y P – Perl). Hay versiónes para Linux, Windows y Mac. Entre las ventajas que ofrece XAMPP están el ser el entorno más utilizado para programar en PHP, gratuito y muy fácil de instalar; además, hay una comunidad muy numerosa detrás en la que puedes apoyarte si tienes problemas.

6.4.1 CÓMO FUNCIONA UNA LLAMADA A UNA PÁGINA PHP

A continuación vamos a explicar gráfica y detalladamente cómo funciona una llamada a una página PHP (por ejemplo *myfpschool.php*):

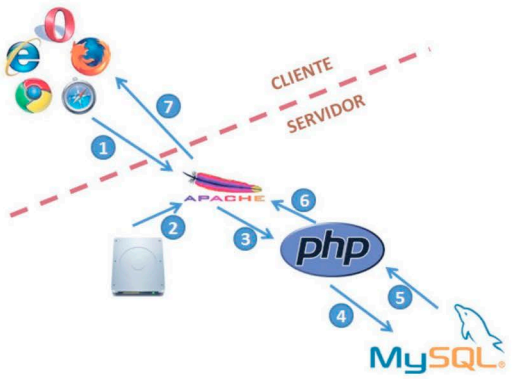


Figura 6.23. *Pasos en la carga de una página PHP*

1 El cliente (navegador, o *browser*) pide al servidor la página web (*myfpschool.php*). Si tenemos instalado el XAMPP, será el servidor web Apache el que recibirá la petición.

2 Una vez recibida la petición, el servidor web localiza la página *myfpschool.php* en su espacio de almacenamiento y la examina por si tuviese código ejecutable de la parte servidor.

3 En caso afirmativo, dado que es una página PHP, la envía al servidor PHP para que la interprete.

4 El servidor PHP interpreta el código y examina si los *scripts* acceden a base de datos. Si el código PHP accede a base de datos, se envían las sentencias SQL al gestor de base de datos (previamente se ha tenido que establecer una conexión con el SGBD).

5 El SGBD ejecuta las sentencias SQL y envía los resultados al servidor PHP.

6 El servidor PHP termina de interpretar el código PHP y le devuelve el resutado HTML al servidor web.

7 El servidor web devuelve el código HTML al cliente (navegador).

6.4.2 TU PRIMER PROGRAMA EN PHP

LOS FICHEROS PHP

Un fichero PHP puede contener texto, HTML, CSS, JavaScript y código PHP.
Como hemos visto en el apartado anterior, el código PHP se ejecuta en la parte servidora y lo devuelve al cliente como código HTML.
Los ficheros tienen extensión ".php" y se localizan dentro del *document root* o carpeta (y subcarpetas) donde almacena el servidor web todas las páginas web.

A continuación se muestra el contenido de nuestra primera página web:

```
<!DOCTYPE html>
<html>
<body>
<?php
echo "¡Hola mundo!";
?>
</body>
</html>
```

Como puedes observar, el código PHP se encuentra dentro de las etiquetas "`<?php`" y "`?>`" y la única sentencia que tiene este *script* es la sentencia `echo "¡Hola mundo!";` que muestra por pantalla con la orden `echo` el contenido del `String` que se le pasa como parámetro.

6.4.3 ALGUNAS DE LAS CARACTERÍSTICAS DE PHP

Los comentarios

Los comentarios en PHP funcionan igual que en Java. `//` para comentar una línea y `/* */` para comentar múltiples líneas.

Mayúsculas/minúsculas

A PHP no le importan las mayúsculas y las minúsculas (no es *case sensitive*), esto quiere decir que las siguientes líneas para PHP son iguales:

```
echo "¡Hola mundo!";
ECHO "¡Hola mundo!";
Echo "¡Hola mundo!";
```

Variables

En PHP no se declaran las variables. Las variables comienzan con el símbolo del dólar `$`. También las variables son *case sensitive* (lo que hace que `$x` y `$X` sean dos variables diferentes). Un ejemplo de utilización de una variable en un *script* sería el siguiente:

```
<?php
$x=5;
echo $x;
?>
```

Tipos de datos en PHP

PHP soporta como tipos de datos los `strings`, `integers`, números en coma flotante, booleanos, `arrays` y objetos. A continuación se muestra un ejemplo de objeto:

```
<?php
class Alumno
{
 var $nombre;
 function Alumno($n="-") {
 $this->nombre = $n;
 }
 function getNombre() {
 return $this->nombre;
 }
}
?>
```

El valor nulo

Como en otro tipo de lenguajes, PHP contempla el valor nulo o `null`, que indica que una variable no tiene ningún valor. También sirve para saber si una variable está vacía o no. Este valor es muy utilizado cuando se trabaja con bases de datos.

En el siguiente ejemplo se puede ver cómo se asigna el valor *null* a una variable:

```
<?php
$varWeb = "myfpschool.com";
$varWeb = null;
```

Las constantes

Las constantes en PHP sí son *case sensitive*, por lo tanto hay que tener cuidado al utilizarlas. Por regla general, los programadores utilizan constantes en mayúsculas siempre. De esa manera no hay problema de equivocación.

Para crear una constante se utiliza la función `define()` y esta función creará la constante que será válida durante todo el *script*. Al contrario que con las variables, no hace falta utilizar el símbolo `$`.

Un ejemplo de constantes sería el siguiente:

```
<?php
define("MIWEB", "myfpschool.com");
echo MIWEB;
```

Los formularios

PHP suele utilizar mucho los formularios `<form>` de HTML para recoger datos de la parte del cliente. Existen dos métodos para recoger datos del navegador del cliente: `GET` y `POST`. Ambos métodos crean un *array* con los valores de los elementos del formulario. La ventaja de `POST` es que los datos no son visibles cuando se invoca la página PHP en el navegador (en la barra de dirección), hace posible añadir la página a favoritos al no tener las variables en la propia dirección y además no tiene el límite de 2000 caracteres que tiene `GET`. Por lo tanto, los programadores prefieren enviar datos por `POST` que por `GET`.

Bases de datos

Con PHP los programas se pueden conectar con cualquier base de datos, de las que MySQL es la más utilizada puesto que también es *open source*, es la más utilizada en la Web, es eficiente tanto para pequeñas y grandes aplicaciones, soporta SQL y está disponible para un gran número de plataformas.

6.5 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Los ficheros HTML también se denominan páginas web y tienen extensión `html` o `htm`.
b) `<head>` es una etiqueta que delimita la cabecera de la página web.
c) La capa de presentación recibe las peticiones de los clientes, las procesa y envía las respuestas.
d) El nuevo estándar HTML5 ofrece un nuevo y potente elemento como son los `<canvas>` o lienzos.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un fichero PHP puede contener texto, HTML, CSS, JavaScript y código PHP.
b) `<!DOCTYPE html>`. Esta etiqueta es una declaración de HTML5 e indica al navegador el tipo de HTML utilizado y la versión.
c) En la capa de presentación residen los datos del sistema.
d) La capa de negocio recibe las peticiones de los clientes de la capa de presentación, las procesa y envía las respuestas.

3 ¿Cuál de las siguientes afirmaciones es falsa?:
- e) Con HTML5 aparece la etiqueta `<video>`, la cual está soportada por casi todos los navegadores.
- a) HTML es un lenguaje de marcas.
- b) Hay que tener cuidado en PHP porque es *case sensitive*, distingue mayúsculas de minúsculas.
- c) PHP soporta como tipos de datos los `strings`, `integers`, números en coma flotante, booleanos, `arrays` y objetos.

4 ¿Cuál de las siguientes afirmaciones es falsa?:
- a) Con la propiedad `border radius` podremos crear cuadros redondeados y con efectos.
- b) En PHP, al contrario que en Java, no existe el valor `null`.
- c) Como en otro tipo de lenguajes, PHP contempla el valor nulo (o `null`) que indica que una variable no tiene ningún valor.
- d) `<strong>` es una etiqueta para definir un texto resaltado.

5 ¿Cuál de las siguientes afirmaciones es falsa?:
- a) La forma más elegante de codificar programas en JavaScript es utilizar funciones fuera de la página web.
- b) Las etiquetas son diferentes si están en mayúsculas o en minúsculas, para HTML `<body>` es diferente de `<BODY>`.
- c) En el año 2012 salió al mercado el último estándar de HTML, que es el HTML5.
- d) Los lenguajes de marcas utilizan etiquetas.

7 Acceso a bases de datos relacionales

La mayoría de los accesos a bases de datos de cualquier aplicación son sobre **bases de datos relacionales**. En el siguiente capítulo se estudiará en profundidad el manejo de bases de datos relacionales con programación orientada a objetos en lenguaje Java.

7.1 BASES DE DATOS RELACIONALES

Una base de datos relacional almacena datos en tablas de tal manera que esos datos puedan ser almacenados y recuperados de una forma eficiente. Las tablas se componen de una serie de objetos o filas (*rows* en inglés) las cuales tienen los mismos elementos.

Un **SGBD** (Sistema Gestor de Bases de Datos) —o lo que es lo mismo DBMS (*DataBase Management System*)— es el proceso responsable de manejar, almacenar y recuperar los datos de una base de datos. En el caso de que la base de datos sea relacional, este proceso se denomina SGBDR (Sistema Gestor de Bases de Datos Relacional), o, lo que es lo mismo, RDBMS (*Relational DataBase Management System*).

Java tiene una API (*Application Programming Interface*) que podemos llamar "librería", la cual permite interactuar con fuentes de datos (incluidas bases de datos) de tal manera que podemos:

- Conectarnos a una fuente de datos (generalmente una base de datos).
- Enviar consultas de selección y actualización de la base de datos.
- Recuperar datos de una consulta y manejarlos.

El producto **JDBC** según Oracle (que es el propietario de esta API y del lenguaje Java) tiene cuatro componentes:

- **La API JDBC**. Ofrece un acceso a bases de datos relacionales desde Java. Con la API de Java se pueden efectuar consultas SQL, recuperar datos de la base de datos y realizar cambios a la base de datos a través del *datasource* (origen de datos). La API JDBC 4.0 está dividida en dos paquetes: `java.sql` y `javax.sql`. Ambos paquetes están incluidos en las plataformas Java SE y Java EE.
- **El administrador de controladores JDBC**. La clase JDBC `DriverManager` define los objetos desde los que se pueden conectar las aplicaciones Java a un controlador JDBC. `DriverManager` ha sido tradicionalmente la columna vertebral de la arquitectura JDBC. Esta clase es pequeña y simple.
- **La suite de test JDBC**. Utilizada para testear las aplicaciones Java que utilizan JDBC.
- **El puente JDBC-ODBC**. El puente software de Java JDBC proporciona acceso a gestores de bases de datos a través de ODBC. Para hacer esto, es necesario tener instalado ODBC en la máquina cliente desde donde se conecta el programa Java.

RECUERDA

En este libro se van a tratar los dos primeros componentes anteriores. Los dos siguientes son menos utilizados (se usan para testear aplicaciones web o para comunicarse con bases de datos vía ODBC).

7.2 DISEÑO DE BASES DE DATOS EN VARIOS NIVELES

Existen dos modelos fundamentales en sistemas informáticos que acceden a bases de datos:

Modelo de arquitectura basada en **dos niveles** (*two-tier*).

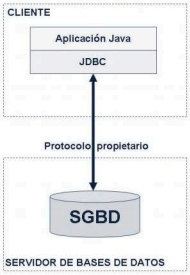


Figura 7.1. *Modelo de arquitectura basado en dos niveles*

En los modelos de arquitectura en dos niveles, el cliente accede directamente a los datos a través del *driver* JDBC.

Los comandos son enviados a la base de datos y los resultados son devueltos a la aplicación cliente. Es una configuración cliente/servidor donde la máquina del usuario que corre la aplicación Java es el cliente y el servidor es donde reside la base de datos. Servidor y cliente pueden estar en máquinas diferentes en la misma red local o a través de Internet.

Modelo de arquitectura basada en **tres niveles** (*three-tier*).

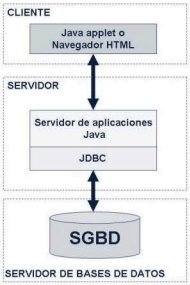


Figura 7.2. *Modelo de arquitectura basado en dos niveles*

En la arquitectura en tres niveles hay una capa intermedia donde reside la lógica del negocio. En esta capa intermedia o *middleware* está situado el servidor de aplicaciones, el cual envía los comandos que recibe de la máquina cliente al gestor de bases de datos y los resultados se los vuelve a enviar al cliente. Esta capa intermedia generalmente suele aportar una mayor seguridad y un acceso más controlado a los datos, con lo que se gana en muchas ocasiones una mejora del rendimiento. Cada vez más se está utilizando Java en la programación de la capa intermedia en detrimento de otros lenguajes de programación como C o C++.

JDBC puede ser implantado sin problemas en cualquiera de estos dos modelos anteriores.

7.3 LENGUAJE DE ACCESO A BASE DE DATOS

Desde 1970 —año en que Codd propuso su modelo relacional— hasta la actualidad, la mayoría de las bases de datos son consultadas mediante lenguaje SQL (*Structured Query Language*). Mediante este lenguaje podremos efectuar consultas en la base de datos y realizar cambios en la misma.

SQL utiliza dos tipos de sublenguajes:

- El **lenguaje de definición de datos** (**LDD**), que permite realizar cambios en la estructura de la base de datos. Órdenes de este lenguaje son `CREATE` (crear), `ALTER` (modificar), `DROP` (eliminar) o `TRUNCATE` (borrar tabla).
- El **lenguaje de manejo de datos** (**LMD**), que permite realizar consultas sobre la base de datos. La orden por excelencia de este lenguaje es `SELECT` (seleccionar). Además de `SELECT` existen otras órdenes como `INSERT` (insertar), `UPDATE` (actualizar) y `DELETE` (borrar).

7.4 API DE ACCESO A LA BASE DE DATOS

Para trabajar con JDBC se necesitará crear un entorno mínimo que permita compilar y ejecutar los programas Java. La manera de desarrollar más cómoda y versátil es tener en la máquina de desarrollo la base de datos y demás software. De esa manera es fácil administrar la base de datos y podremos evitar todos los problemas que pudieran surgir al tener el cliente y la base de datos en máquinas distintas.

Para crear el entorno JDBC se deberá tener lo siguiente:

- Una versión de **Java** (preferiblemente la última versión del Java SE SDK).
- Por supuesto, una **base de datos**. En todos los ejemplos siguientes se va a emplear MySQL. MySQL es una base de datos relacional, multihilo y multiusuario. Esta base de datos está licenciada de forma dual (software libre y propietario). Solamente si se desea incorporar esta base de datos a software propietario es necesario licenciar el producto.
- Los **drivers** necesarios para conectarse con la base de datos utilizada. En el caso de que se utilice MySQL como base de datos, habrá que instalar Connector/J (preferiblemente la última versión). Para instalar estos *drivers* necesitarás modificar la variable de entorno `CLASSPATH` y situar el fichero `JAR` en su ubicación correcta.

CONSEJO

Yo, en vez de instalar solo MySQL, he decidido instalar XAMPP versión Lite (software libre gratuito) y de esa manera instalo Apache, MySQL, PHP y PhpMyAdmin. Eso me permite administrar y manejar la base de datos desde un navegador web.

7.5 NIVEL APLICACIÓN

A partir de este apartado profundizaremos más en cómo Java trabaja con bases de datos y veremos qué comandos, métodos y clases hay que utilizar para que nuestros programas accedan a una base de datos relacional, que, a la postre, son las más utilizadas.

7.5.1 ESTABLECIMIENTO DE UNA CONEXIÓN CON UNA BASE DE DATOS

Antes de trabajar con la base de datos hay que establecer una conexión con la misma. JDBC se conecta a las bases de datos utilizando una de estas dos clases:

- **`DriverManager`**. Es la forma más sencilla de conectarse a una base de datos. La conexión con la base de datos se realiza especificando una dirección URL.
- **`DataSource`**. Esta clase hace que los detalles sobre la base de datos a la que se conecta sean transparentes a la aplicación.

En los ejemplos que se van a ver a continuación se va a utilizar la clase `DriverManager` para establecer conexiones con la base de datos.

RECUERDA

No olvides importar el paquete `java.sql` cuando tu programa utilice JDBC. Recuerda ejecutar la siguiente sentencia al inicio de tu programa `.java`:

```
import java.sql.*;
```

El siguiente código permite realizar una conexión con una base de datos MySQL:

```
try {
connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/
test","andrés","pelusilla");
System.out.println("Connection succeed!");
}
catch (Exception e) {
e.printStackTrace();
}
```

Nótese que se utiliza el método `getConnection` de la clase DriverManager. Para establecer la conexión se pasan tres parámetros a este método:

- La **URL**: `"jdbc:mysql://localhost:3306/test"`, donde:
  - **`localhost`**. Es la dirección de la máquina donde reside la base de datos.
  - **`3306`**: Es el puerto donde escucha la base de datos.
  - **`test`**: Es la base de datos a la que se conectará el programa.
- El **usuario** con el que se conecta a la base de datos: `"andres"`.
- La **password** del anterior usuario: `"pelusilla"`.

7.6 MANEJANDO SQLEXCEPTIONS

Cuando JDBC encuentra un error al trabajar con una base de datos, en vez de una `Exception` lanza una `SQLException`. A la hora de programar, el objeto `SQLException` contiene mucha información que puede servirnos de ayuda para determinar el origen del error:

Descripción del error. Se puede recuperar esta descripción utilizando el método `SQLException.getMessage`, el cual devuelve un dato de tipo `String`.

- **Código SQLState**. Estos códigos y su significado están estandarizados por la ISO/ANSI y el Open Group. Este código es un objeto `String` y se puede recuperar mediante el método `SQLException.getSQLState`.
- **Código de error**. Código numérico (`integer`) que identifica el error producido. Este código se puede obtener llamando al método `SQLException.getErrorCode`.
- **Causa del error**. Una `SQLException` puede haber sido lanzada debido a una o varias causas. Para recuperar todas estas causas basta con llamar de manera recursiva al método `SQLException.getCause` hasta que se recupere el valor `null`.
- Si en vez de una sola excepción se han producido varias, se pueden recuperar llamando al método `SQLException.getNextException` en la excepción lanzada.

El siguiente código muestra la manera de tratar una excepción lanzada por el programa:

```
try {
.....
} catch (SQLException e ) {
        printSQLException(e);
} finally {
.....
}
```

Como se puede observar, ahora se maneja una `SQLException` en vez de una `Exception` como anteriormente se hacía. El tratamiento de excepciones se realiza en la función `printSQLException`, la cual se detalla a continuación:

```
public static void printSQLException(SQLException ex) {
ex.printStackTrace(System.err);
System.err.println("SQLState: " + ex.getSQLState());
System.err.println("Error Code: " + ex.getErrorCode());
System.err.println("Message: " + ex.getMessage());
Throwable t = ex.getCause();
while(t != null) {
System.out.println("Cause: " + t);
t = t.getCause();
}
}
```

En esta función se ve cómo se muestra por pantalla; además del mensaje que muestra Java una vez producido el error, mostrará el código SQLState (`getSQLState()`), el código de error (`getErrorCode()`) y el mensaje de error (`getMessage()`). Todos estos métodos corresponden a la instancia del objeto `SQLException`. Nótese también que con el método `getCause()` se van recorriendo las diferentes causas del error producido. En el momento que `getCause()` devuelve `null` es que no existen más causas del error.

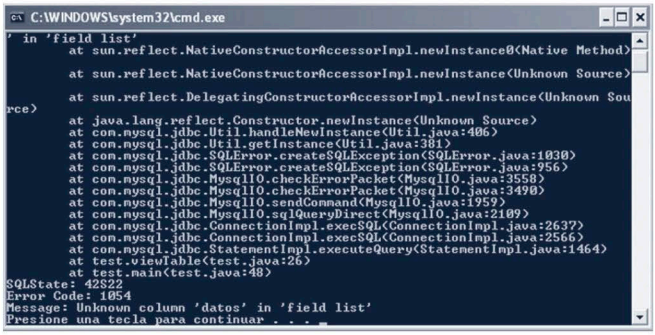


Figura 7.3. *Excepción lanzada por el programa*

La figura anterior muestra el resultado del tratamiento de una excepción Java.

7.7 CREACIÓN Y CARGA DE DATOS EN TABLAS

Para los siguientes ejemplos se va a crear la siguiente estructura de tablas.

Figura 7.4. *Relación entre las tablas equipo y jugadores*

En esta estructura existen dos tablas, la de los equipos y la de los jugadores. Cada jugador tiene un equipo por el que juega y un equipo se compone de varios jugadores.

Las sentencias de creación de estas dos tablas son las siguientes:

```
create table EQUIPO
(TEAM_ID integer NOT NULL,
EQ_NOMBRE varchar(40) NOT NULL,
ESTADIO varchar(40) NOT NULL,
POBLACION varchar(20) NOT NULL,
PROVINCIA varchar(20) NOT NULL,
COD_POSTAL char(5),
PRIMARY KEY (TEAM_ID));

create table JUGADORES
(PLAYER_ID integer NOT NULL,
TEAM_ID integer NOT NULL,
NOMBRE varchar(40) NOT NULL,
DORSAL integer NOT NULL,
EDAD integer NOT NULL,
PRIMARY KEY (PLAYER_ID),
FOREIGN KEY (TEAM_ID) REFERENCES EQUIPO (TEAM_ID));
```

7.7.1 CREACIÓN DE TABLAS CON JDBC

Una vez tenemos estas sentencias de creación, las incorporamos a un método de la clase, el cual nos va a ayudar a crear las dos tablas:

```
public static void createEQUIPO(Connection con, String BDNombre) throws SQLException {
 String createString = "create table " + BDNombre + ".EQUIPO " +
                  "(TEAM_ID integer NOT NULL," +
                  "EQ_NOMBRE varchar(40) NOT NULL," +
                  "ESTADIO varchar(40) NOT NULL," +
                  "POBLACION varchar(20) NOT NULL," +
                  "PROVINCIA varchar(20) NOT NULL," +
                  "COD_POSTAL char(5)," +
                  "PRIMARY KEY (TEAM_ID))";
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 stmt.executeUpdate(createString);
 } catch (SQLException e) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
}
public static void createJUGADORES(Connection con, String BDNombre) throws SQLException
{
 String createString = "create table " + BDNombre + ".JUGADORES" +
                   "(PLAYER_ID integer NOT NULL," +
                   "TEAM_ID integer NOT NULL," +
                   "NOMBRE varchar(40) NOT NULL," +
                   "DORSAL integer NOT NULL," +
                   "EDAD integer NOT NULL," +
                   "PRIMARY KEY (PLAYER_ID)," +
                   "FOREIGN KEY (TEAM_ID) REFERENCES EQUIPO (TEAM_ID))";
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 stmt.executeUpdate(createString);
 } catch (SQLException e) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
}
```

Al ejecutar este método se crearán las tablas necesarias para tratar con los ejemplos siguientes. Es necesario crear primero la tabla *Equipo* y luego la tabla *Jugadores*, pues depende de la primera.

LA CLASE STATEMENT

El objeto `stmt` anterior de la clase `Statement` lo hemos utilizado para enviar sentencias SQL a la base de datos. Existen tres tipos de objetos `Statement`:

`Statement`. Como se ha visto, servirá para enviar órdenes SQL a la base de datos sin parámetros.

`PreparedStatement`. Hereda de `Statement`. Se utiliza para ejecutar comandos SQL con o sin parámetros de entrada ya precompilados.

`CallableStatement`. Hereda de `PreparedStatement`. Se utiliza para llamar a procedimientos almacenados de base de datos. Permite trabajar con parámetros de entrada y de salida.

Un objeto de la clase `Statement` se crea mediante el método de `Connection createStatement`. Con el objeto `Statement` se pueden ejecutar comandos SQL y recibir los resultados.

Para crear un objeto `statement` se utiliza el método `createStatement()` de un objeto de tipo `Connection`. Para crear el objeto de manera exitosa primero hay que conectarse a la base de datos. En el siguiente código se puede ver cómo se realizaría:

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/
test","root","");
Statement stmt = con.createStatement();
```

Una vez realizada la conexión y creado el objeto `Statement`, se puede llamar a tres métodos diferentes para ejecutar sentencias SQL:

`executeQuery`. Se utiliza para ejecutar sentencias `SELECT` y la llamada a este método devuelve un `resultset` que es un objeto para poder tratar los datos devueltos por la base de datos.

`executeUpdate`. Como se puede ver en el ejemplo anterior, se puede utilizar para ejecutar sentencias DDL (*Data Definition Language* - Lenguaje de definición de datos) como `Create Table` o `Drop Table`. No obstante, se puede utilizar para ejecutar sentencias `Insert`, `Update` y `Delete`, las cuales son más utilizadas en aplicaciones que las primeras. Este método devuelve un entero que indica el número de filas afectadas por la sentencia (en sentencias DDL siempre es 0).

`execute`. Utilizado en sentencias que devuelven más de un `resultset`. Se utiliza solamente en programación avanzada.

Como buena práctica, se recomienda cerrar los objetos `statement` mediante este comando:

```
stmt.close();
```

No obstante, los objetos `Statement` se cierran automáticamente por el *garbage collector* de Java (recolector de basura). La llamada al método `close()` hace que se libere inmediatamente la basura y se eviten posibles problemas con la memoria.

7.7.2 CARGA DE DATOS EN LAS TABLAS CON JDBC

Los siguientes métodos son los encargados de cargar los datos en las tablas de equipos y jugadores. La estructura es prácticamente igual a las anteriormente vistas de creación de las tablas, lo único que cambia es la sentencia SQL que se ejecuta.

```
public static void cargaEQUIPO(Connection con, String BDNombre) throws SQLException {
 Statement stmt = null;
 try {
```

```
 stmt = con.createStatement();
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".EQUIPO VALUES ("
                    +"1,'ESTEPONA','MONTERROSO','ESTEPONA','MALAGA','29680')");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".EQUIPO VALUES ("
                    +"2,'ALCORCON','SANTO DOMINGO','ALCORCON','MADRID','28924')");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".EQUIPO VALUES ("
                    +"3,'PORCUNA','SAN CRISTOBAL','PORCUNA','JAEN','23790')");
 } catch (SQLException e) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
}
public static void cargaJUGADORES(Connection con, String BDNombre) throws SQLException {
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 //Cargando datos de Estepona
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"1,1,'JOSE ANTONIO',1,42)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"2,1,'IGNACIO',2,62)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"3,1,'DIEGO',3,20)");
     //Cargando datos de Alcorcón
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"4,2,'TURRION',1,37)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"5,2,'LUIS ABEL',2,37)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"6,2,'ISAAC',3,40)");
     //Cargando datos de Porcuna
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"7,3,'JUAN FRANCISCO',1,33)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"8,3,'PARRA',2,37)");
 stmt.executeUpdate("INSERT INTO " + BDNombre + ".JUGADORES VALUES ("
                    +"9,3,'RAUL',3,19)");
 } catch (SQLException e) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
}
```

7.8 RECUPERAR LA INFORMACIÓN DE LA BASE DE DATOS

Para recuperar la información de la tabla *Equipos* se puede utilizar el método que se describe a continuación:

```
public static void verEQUIPO(Connection con, String BDNombre) throws SQLException {
 Statement stmt = null;
 String query = "select EQ_NOMBRE ,ESTADIO ,POBLACION ,PROVINCIA "+
                              " from " + BDNombre + ".EQUIPO";
 try {
 stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery(query);
 while (rs.next()) {
 String equipo = rs.getString("EQ_NOMBRE");
 System.out.println("Equipo: "+equipo);
 String estadio = rs.getString("ESTADIO");
 System.out.println("Equipo: "+estadio);
 String poblacion = rs.getString("POBLACION");
 System.out.println("Equipo: "+poblacion);
 String provincia = rs.getString("PROVINCIA");
 System.out.println("Equipo: "+provincia);
 System.out.println("***********************************************");
 }
 } catch (SQLException e ) {
          printSQLException(e);
 } finally {
 stmt.close();
 }
}
```

En el método anterior se puede observar que se utiliza el objeto `ResultSet`, el cual representa un conjunto de datos recuperado de una base de datos (matriz de datos). En este procedimiento se crea un objeto `ResultSet` llamado `rs` que recibe la información cuando se ejecuta la consulta SQL mediante el objeto `stmt` de la clase `Statement`.

Se pueden crear objetos `ResulSet` a partir del cualquier objeto que implemente la interfaz `Statement`, como, por ejemplo, `PreparedStatement`, `CallableStatement` y `Rowset`.

RECUERDA

El acceso a los datos mediante el `Resultset` se denomina "cursor". No confundas este cursor con los cursores de bases de datos. Son dos cosas diferentes. El cursor del que estamos hablando es un puntero a una zona de memoria donde residen los datos recuperados por el comando `SQL`. Inicialmente se coloca en una posición anterior a la primera posición de los datos recuperados y mediante la llamada al método `ResultSet.next()` vamos posicionándonos en la siguiente fila de los datos recuperados. Esto se suele hacer utilizando un bucle. Al final del bucle, cuando ya no existen más datos, el método `next()` devuelve `false`.

7.8.1 LA INTERFAZ RESULTSET

La interfaz `Resultset`, como hemos visto, tiene métodos para recuperar y manipular los datos relativos a comandos `SQL` realizados a una base de datos. Existen distintos tipos de objetos `ResultSet` dependiendo de sus características.

Tipos de `ResultSet`:

- **`TYPE_FORWARD_ONLY`**. Este cursor es el cursor por defecto. Los ejemplos anteriores están realizados con este cursor. Como su nombre indica, es un cursor unidireccional y solo se mueve en un sentido hacia delante (desde la primera fila hasta la última).
- **`TYPE_SCROLL_INSENSITIVE`**. Este cursor puede moverse hacia delante y hacia atrás (*forward* y *backward*) siempre teniendo en cuenta la posición en la que se encuentra el cursor. Aunque los datos con los que está trabajando cambien en la base de datos no le afectará. Contiene los datos que se recuperaron cuando se ejecutó el comando `SQL`.
- **`TYPE_SCROLL_INSENSITIVE`**. Este cursor, al igual que el anterior, puede moverse hacia delante y hacia atrás, la diferencia radica en que cuando los datos con los que está trabajando cambian, en la base de datos el cursor al moverse trabaja con los datos más actuales reflejando los últimos cambios realizados.

Concurrencia:

Determina si los datos del `ResultSet` son actualizables en la base de datos o no. Existen dos niveles de concurrencia:

- **`CONCUR_READ_ONLY`**. Es el tipo de concurrencia por defecto. El objeto `ResultSet` NO puede ser actualizado utilizando la interfaz `ResultSet`.
- **`CONCUR_UPDATABLE`**. El objeto `ResultSet` NO puede ser actualizado utilizando la interfaz `ResultSet`.

No todos los *drivers* JDBC soportan la concurrencia con base de datos. El método `DatabaseMetaData.supportsResultSetConcurrency` devolverá `true` (verdadero) si el nivel de concurrencia es soportado por el *driver*, y falso en caso contrario.

7.8.1.1 Persistencia

Cuando se llama al método `Connection.commit` esto puede implicar que los objetos `ResultSet` que estaban abiertos en la transacción se cierren. Esto puede provocar errores en el programa. Mediante la propiedad `holdability` del cursor se puede especificar el funcionamiento del cursor cuando se ejecuta un `commit`.

Cuando se llama a los métodos `createStatement`, `prepareStatement`, y `prepareCall` del objeto `Connection`, se le pueden pasar las siguientes constantes:

- **`HOLD_CURSORS_OVER_COMMIT`**. Los cursores `ResultSet cursors` NO se cerrarán cuando se ejecuta el método `commit`.
- **`CLOSE_CURSORS_AT_COMMIT`**. Los cursores `ResultSet cursors` SÍ se cerrarán cuando se ejecuta el método `commit`.

El tipo de persistencia varía dependiendo del gestor de base de datos. Algunas bases de datos no soportan alguno de estos tipos de persistencia.

7.8.2 OTRA MANERA DE RECUPERAR LOS DATOS DE UNA TABLA

Existe otra manera diferente de recuperar los datos de un `Resultset`, en vez de utilizar los nombres de los campos en los métodos `getString`, `getBoolean`, `getLong`, etc. La solución es llamar a los campos por el orden que ocupan en la sentencia `SQL`, comenzando por el número 1.

```
public static void verEQUIPO(Connection con, String BDNombre) throws SQLException {
 Statement stmt = null;
 String query = "select EQ_NOMBRE ,ESTADIO ,POBLACION ,PROVINCIA "+
                              " from " + BDNombre + ".EQUIPO";
 try {
 stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery(query);
 while (rs.next()) {
 String equipo = rs.getString(1);
 System.out.println("Equipo: "+equipo);
 String estadio = rs.getString(2);
 System.out.println("Equipo: "+estadio);
 String poblacion = rs.getString(3);
 System.out.println("Equipo: "+poblacion);
 String provincia = rs.getString(4);
 System.out.println("Equipo: "+provincia);
 System.out.println("***********************************************");
 }
 } catch (SQLException e ) {
          printSQLException(e);
 } finally {
 stmt.close();
 }
}
```

Este tipo de recuperación de información se utiliza cuando recuperamos varias columnas que tienen el mismo nombre. De la manera anterior no sería posible recuperar la información, y de esta manera sí.

RECUERDA

El método `getString()` puede servirnos para recuperar datos `CHAR` y `VARCHAR` de las bases de datos. También es posible recuperar datos numéricos de la base de datos, pero hay que tener en cuenta que estos serán convertidos a `String`.

7.8.3 LOS CURSORES

Los cursores en JDBC son los anteriormente vistos `ResulSet`. Cuando se crea un `ResulSet`, este se coloca antes de la primera fila de datos. Ya hemos visto cómo los cursores por defecto son unidireccionales y solo se mueven hacia

delante. No obstante, se pueden crear cursores bidireccionales que pueden utilizar otros métodos para desplazarse por los datos como son:

- **next()**. Mueve el cursor una posición hacia delante. Devuelve `true` si el cursor está posicionado en una fila y `false` en caso de que esté después de la última fila. Es el único método que se puede llamar cuando se crea un cursor por defecto (`TYPE_FORWARD_ONLY`).
- **previous()**. Mueve el cursor una posición hacia atrás. Devuelve `true` si el cursor está posicionado en una fila y `false` en caso de que esté antes de la primera fila.
- **first()**. Coloca el cursor en la primera fila. Devuelve `true` si el cursor contiene al menos una fila y `false` en caso contrario.
- **last()**. Coloca el cursor en la última fila. Devuelve `true` si el cursor contiene al menos una fila y `false` en caso contrario.
- **beforeFirst()**. Coloca el cursor antes de la primera fila.
- **afterLast()**. Coloca el cursor después de la primera fila.
- **relative(int rows)**. Mueve el cursor `rows` de forma `relative` a la actual posición.
- **absolute(int row)**. Coloca el cursor en la posición especificada en el parámetro `row`.

7.9 MODIFICACIÓN Y ACTUALIZACIÓN DE LA BASE DE DATOS

En este apartado veremos cómo se modifican y actualizan datos en la base de datos utilizando la sintaxis clásica y los objetos `Resulset`.

7.9.1 MODIFICACIÓN CLÁSICA DE DATOS

La modificación de una tabla en una base de datos es similar a la ejecución de otras sentencias como las inserciones y borrados en tablas. Únicamente cambia la sintaxis SQL. El siguiente método muestra un procedimiento básico de actualización de una columna en una base de datos:

```
public static void modificaEQUIPO(Connection con, String BDNombre) throws SQLException {
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 stmt.executeUpdate("UPDATE " + BDNombre + ".EQUIPO SET ESTADIO = 'ALBORAN' "+
 " WHERE TEAM_ID = 1");
 } catch (SQLException e) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
}
```

7.9.2 MODIFICACIÓN DE DATOS EN LAS TABLAS UTILIZANDO RESULTSET

Como se puede ver en el siguiente ejemplo, en Java se pueden crear `ResultSet` bidireccionales y actualizables. El siguiente método actualiza la edad de los jugadores y le suma el valor introducido en el parámetro entero `cuantoMas`.

```
public static void modificaEdadJugadores(Connection con, String BDNombre,int cuantoMas)
throws SQLException {
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
 ResultSet rs = stmt.executeQuery(
 "SELECT * FROM " + BDNombre + ".JUGADORES");

 while (rs.next()) {
 int i = rs.getInt("EDAD");
 rs.updateInt("EDAD", i + cuantoMas);
 rs.updateRow();
 }

 } catch (SQLException e ) {
 printSQLException(e);
 } finally {
 stmt.close();
 }
 }
```

Como se estudió anteriormente, la propiedad `TYPE_SCROLL_SENSITIVE` hace que el objeto `ResultSet` creado pueda moverse bidireccionalmente de forma relativa a su posición actual; la propiedad `CONCUR_UPDATABLE` hace que se puedan modificar los datos del cursor y estos se repliquen a la base de datos. Hasta que no se invoca al método `ResultSet.updateRow`, no se actualizará la base de datos.

7.9.3 INSERTAR DATOS EN LAS TABLAS UTILIZANDO RESULTSET

```
public static void insertaJUGADOR(Connection con, String BDNombre, int player_id,int
team_id,String nombre,int dorsal,int edad)
 throws SQLException {
 Statement stmt = null;
 try {
 stmt = con.createStatement();
 stmt = con.createStatement(
 ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
 ResultSet rs = stmt.executeQuery(
 "SELECT * FROM " + BDNombre + ".JUGADORES");

 rs.moveToInsertRow();
```

```
    rs.updateInt("PLAYER_ID", player_id);
    rs.updateInt("TEAM_ID", team_id);
    rs.updateString("NOMBRE", nombre);
    rs.updateInt("DORSAL", dorsal);
    rs.updateInt("EDAD", edad);

    rs.insertRow();
    rs.beforeFirst();

    } catch (SQLException e ) {
    printSQLException(e);
    } finally {
    stmt.close();
    }
  }
```

Es posible que el *driver* JDBC utilizado no tenga la posibilidad de insertar datos en la base de datos. En ese caso, se lanza la excepción `SQLFeatureNotSupportedException`. Al ejecutar el método `insertRow()` del objeto `ResultSet` se insertarán los datos tanto en el `ResultSet` como en la base de datos.

7.10 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un SGBD es lo mismo que un DBMS.
b) La API JDBC 4.0 está dividida en dos paquetes: `java.sql` y `javax.sql`.
c) Para trabajar con JDBC se necesitará crear un entorno mínimo que permita compilar y ejecutar los programas Java.
d) Los objetos `statement` no se cierran automáticamente, se recomienda cerrar los objetos `statement` mediante el comando `stmt.close();`.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Una `SQLException` puede haber sido lanzada debido a una o varias causas.
b) Una base de datos relacional almacena datos en tablas de tal manera que esos datos puedan ser almacenados y recuperados de una forma eficiente.
c) Las tablas se componen de una serie de objetos o filas (*rows* en inglés), las cuales tienen los mismos o distintos elementos.
d) Un SGBDR es lo mismo que un RDBMS.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El puente software de Java JDBC proporciona acceso a gestores de bases de datos a través de ODBC.

b) En los modelos de arquitectura en tres niveles, el cliente accede directamente a los datos a través del *driver* JDBC.

c) La clase JDBC `DriverManager` define los objetos desde los que se pueden conectar las aplicaciones Java a un controlador JDBC.

d) En la URL "`jdbc:mysql://localhost:3306/test`", `localhost` es la dirección de la máquina donde reside la base de datos.

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) En la URL "`jdbc:mysql://localhost:3306/test`", `localhost` es la base de datos a la que se conectará el programa.

b) En la arquitectura en tres niveles hay una capa intermedia donde reside la lógica del negocio.

c) `PreparedStatement` se utiliza para ejecutar comandos SQL con o sin parámetros de entrada ya precompilados.

d) La API JDBC ofrece un acceso a bases de datos relacionales desde Java.

8 Calidad, pruebas, documentación y proceso de ingeniería del software

Calidad del software, pruebas, documentación, ingeniería del software, etc., son conceptos que todo programador tiene que dominar si quiere adquirir un nivel aceptable y progresar. En este capítulo se profundizará en estos conceptos.

8.1 CALIDAD EN EL DESARROLLO DEL SOFTWARE

El origen del interés actual por la calidad se puede explicar recurriendo al estudio de la evolución en la comercialización de los productos. En un mercado tan competitivo como el de hoy no basta con producir y distribuir masivamente los productos o servicios: vender es lo importante y solo se consigue con la seguridad de la aceptación del cliente.

La calidad del software se convierte así en un **objetivo fundamental** para las empresas junto a los dos parámetros clásicos de su gestión: dinero y tiempo. Lo que prima es la adaptación a las necesidades del cliente y esto lleva a investigar primero cuáles son esas necesidades (investigación de mercados) para luego definirlas en forma de requisitos que se han de cumplir (especificaciones).

Las características específicas del software convierten la tarea de garantizar su calidad en algo mucho más difícil que, por ejemplo, el desarrollo de un producto a través de un proceso industrial.

Así pues, la **calidad es un parámetro fundamental** que permite obtener una idea sobre su adecuación a los fines para los que ha sido construido; y más en el mundo actual, en el que las aplicaciones informáticas se están introduciendo en todos los ámbitos, circunstancia que hace imprescindible que esté libre de defectos y errores.

Los años noventa fueron escenario de una gran **crisis del software**, cuyas principales características fueron:

- Calidad insuficiente del producto final.
- Estimaciones de duración de proyectos y asignación de recursos inexactas, con el problema que ello conlleva.
- Escasez de personal cualificado en un mercado laboral de alta demanda.
- Tendencia al crecimiento del volumen y complejidad de los productos.

Con el tiempo se ha constatado que la calidad no se mide solamente por unos parámetros de funcionamiento, hay otros aspectos que son importantes: el soporte, es decir, el respaldo organizacional que tiene un producto, la formación, la asistencia ante problemas inesperados y el mantenimiento permanente y efectivo.

Para evaluar dicha calidad se lleva a cabo la **evaluación** y **rendimiento de las aplicaciones**.

Las mediciones de rendimiento de un software pueden estar **orientadas hacia el usuario** (ej. tiempos de respuesta) u **orientadas hacia el sistema** (ej. uso de la CPU). Son medidas típicas del rendimiento diferentes variables de tiempo (tiempo de retorno, tiempo de respuesta, tiempo de reacción), la capacidad de ejecución, la carga de trabajo, la utilización, etc.

Para evaluar el software es necesario contar con criterios adecuados que nos permitan analizarlo desde diferentes puntos de vista.

Las **pruebas de rendimiento** son las que se llevan a cabo para determinar lo rápido que realiza una tarea un software en condiciones particulares de trabajo. Sirven también para evaluar otros atributos de la calidad del software como son la **escalabilidad**, la **fiabilidad** y el **uso de los recursos**.

En ocasiones se emplean las **pruebas de carga**, que observan el comportamiento de una aplicación bajo una cantidad de peticiones esperada –como un número elevado de usuarios concurrentes usando la aplicación, que realizan un número elevado de transaccione– y muestran los tiempos de respuesta de las transacciones.

Y también son útiles las **pruebas de estrés** que van doblando el número de usuarios que manejan la aplicación hasta que se rompe, para determinar la solidez de la aplicación en momentos de carga extrema.

Existen otras pruebas, como la **prueba de estabilidad**, que somete la aplicación a una carga continuada; y las **pruebas de picos**, donde la carga va cambiando.

Un *benchmark* es una aplicación o conjunto de aplicaciones cuya finalidad es evaluar el rendimiento de un sistema. Existen cuatro categorías generales de pruebas de comparación:

- **Pruebas de aplicaciones-base**, que se encargan de ejecutar y cronometrar los tiempos de las mismas.
- **Pruebas playback**, que usan llamadas al sistema durante actividades específicas de una aplicación como uso de disco o llamadas a rutinas de gráficos, ejecutándolas aisladamente.
- **Pruebas sintéticas**, que enlazan actividades de la aplicación en subsistemas específicos.
- **Pruebas de inspección**, que no intentan imitar la actividad sino que las ejecuta directamente en su entorno productivo.

Existen varios programas para llevar a cabo este tipo de pruebas, como Winstone o Winbench de ZDNet.

8.1.1 CRITERIOS O FACTORES DE CALIDAD

Los criterios o factores de calidad de un software se definen al principio del proyecto y se siguen durante toda la vida del mismo. Dichos criterios deben poder medirse; si no, no tiene lógica establecerlos. Algunos criterios se pueden **medir directamente**, como podría ser el número de errores que pueda contener un programa, y otros habrá que medirlos de forma **indirecta** (por ejemplo, la facilidad de realizar cambios a los programas).

Generalmente, para evaluar los criterios de calidad se realizan **RTF** o **revisiones técnicas formales**.

LA REVISIÓN TÉCNICA FORMAL O RTF

Las RTF son una herramienta que permite garantizar la calidad, dado que se han demostrado efectivas a la hora de descubrir defectos y comprobar que se han seguido los criterios de calidad previstos.

8.1.2 MÉTRICAS Y ESTÁNDARES DE CALIDAD

Como ya se ha explicado, para evaluar los criterios o factores de calidad se establecen métricas. Algunas métricas de calidad utilizadas son las siguientes:

- **Facilidad de expansión**. Facilidad que tiene un software de poder añadir nuevas funcionalidades.
- **Independencia del hardware**. Capacidad de un software para poder ejecutarse en múltiples plataformas (Java es uno de los lenguajes más independientes de la plataforma que existen).
- **Modularidad**. Número de componentes independientes de un programa.
- **Estandarización de los datos**. Si se utilizan estructuras de datos estándar a lo largo de un programa.
- **Tolerancia a errores**. Efectos que tiene un error en el software.

8.1.3 NORMAS DE CALIDAD Y ESTILO

Generalmente, cuando creamos un programa debemos seguir unas normas de calidad, o, mejor dicho, de estilo de programación. A continuación se dará un resumen muy reducido de normas de estilo que generalmente se siguen cuando se programa en el lenguaje Java:

No tomes estas reglas al pie de la letra. Lo más importante cuando se escribe un programa es que sea fácilmente entendible y tenga una estructura lógica y coherente. Siguiendo estas reglas se pueden escribir programas complicados o imposibles de entender. Utiliza el sentido común.

- Nombres.
  - Las variables deben comenzar con minúsculas.

    ```
    linea
    ```

  - Las variables compuestas deben escribirse combinando mayúsculas y minúsculas.

    ```
    lineaDocumento
    ```

  - Las constantes siempre van en mayúsculas.

    ```
    MAX_LINEAS
    ```

  - Los nombres de procedimientos y métodos deben escribirse combinando mayúsculas y minúsculas.

    ```
    calcularTotal()
    ```

  - Los iteradores, o variables que recorren una serie de valores, deberían nombrarse `i`, `j`, `k`, `l`, etc.

    ```
    for (int i = 0; i <nTables; i++) {
    :
    }
    ```

- Líneas muy largas.

  Una buena solución es dividir las líneas muy largas en partes para hacerlas más legibles en pantalla.

  ```
  System.out.println("Esta es una línea muy larga que" +
  "la he dividido en dos partes.");
  ```

- Arrays.

  Colocar los *brackets* al lado del tipo.

  ```
  double[] lista; // NO: double lista[];
  ```

- Bucles.
  - Utiliza siempre `while` en vez de `do while`. Los los programas son más legibles siempre con `while` porque la condición está al principio y no al final. Además, los `do-while` no son necesarios, siempre se puede cambiar un `while` por un `do-while`. Y por último, si reduces el número de estructuras, los programas siempre serán más homogéneos y sencillos de entender.
  - Evitar el uso de *break* en bucles.
- Sentencias condicionales.

  Los `if` e `if -else` deberían tener este aspecto:

```
if (condiciones) {
sentencias;
}

if (condiciones) {
sentencias;
} else {
sentencias;
}
```

- Uso de espacios en blanco.

  En muchas sentencias, como, por ejemplo, con los operadores es más legible introducir espacios en blanco entre ellos:

```
a = (b + c) * d; // NO: a=(b+c)*d;
```

8.2 PRUEBAS

Los proyectos de desarrollo de software han sufrido tradicionalmente problemas de calidad, tanto en el propio proceso de desarrollo como en los productos que entregan. Estos problemas surgen habitualmente en las desviaciones de plazos y esfuerzo sobre los valores previstos, y en la aparición de fallos durante la implantación y mantenimiento de dichos productos.

Las **pruebas de software** o *testing* son aquel conjunto de procesos que permite **verificar y validar** la calidad de un producto software identificando errores de diseño e implementación (el programa no hace exactamente lo que se pedía; o lo hace, pero de forma incorrecta).

Este tipo de pruebas se encargan de ejecutar el software que se está desarrollando o ya está desarrollado bajo condiciones controladas, y de aplicar sobre el mismo un conjunto de herramientas, técnicas y métodos para tratar de descubrir qué errores tiene. Dichas condiciones controladas pueden ser normales o anormales, tratando intencionadamente de forzar al programa y producir errores en las respuestas para determinar si ocurren sucesos cuando no tendrían que ocurrir, o viceversa.

Se pretende detectar **errores de programación** o *bugs* (fallos en la semántica del código de la aplicación en el lenguaje en que se programase) y lo que se denominan **defectos de forma** (que el programa no realizase lo que el usuario espera).

8.2.1 TIPOS DE PRUEBAS

Existen muchos tipos de pruebas dependiendo del tipo de comprobación que se lleve a cabo. Básicamente, se efectúan dos tipos de comprobaciones:

- **Verificación**. Consiste en demostrar que un programa cumple con sus especificaciones. Se centra en la comprobación de las distintas fases del desarrollo antes de pasar a la siguiente.

  La verificación incluye por parte de los desarrolladores la revisión de los planes, del código, de los requerimientos, de la documentación y de las especificaciones, y, posteriormente, una reunión con los usuarios para evaluar dichos documentos.

  Se trata de dar respuesta a la pregunta: "¿Está el producto correctamente construido?".

  Esto se lleva a cabo mediante listas de comprobaciones (*checking list*), listas de problemas, inspecciones (reuniones formales entre miembros del equipo de desarrollo de diferentes etapas) y *walkthrough* (reunión informal entre analistas y usuarios para la evaluación de propuestas informacionales).

- **Validación**. Se encarga de comprobar que el programa da la respuesta que espera el usuario. Se centra en la comprobación de los requerimientos del software.

  Se trata de dar respuesta a la pregunta: "¿El producto construido es correcto?".

  La validación incluye las pruebas del software y comienza después de que la verificación esté completa.

Existen innumerables tipos de pruebas, entre las que podemos nombrar las siguientes: pruebas de caja negra o caja blanca, unidad de testeo o prueba, integración incremental, pruebas de integración, prueba de interfaces, prueba funcional, prueba de sistema, prueba de fin a fin, prueba de sanidad, prueba de regresión, prueba de aceptación, prueba de carga, prueba de estrés, prueba de performance, prueba de instalación y desinstalación, prueba de recuperación, prueba de seguridad, prueba de compatibilidad, prueba de exploración, prueba de anuncio, prueba de comparación, prueba alfa, prueba beta, prueba de mutación.

Pasemos a describir en profundidad algunas de las pruebas más conocidas:

- **Prueba de caja negra y caja blanca**. En este tipo de pruebas solo se tienen en cuenta las entradas y salidas de la aplicación o sistema. Lo que se va a testear en este tipo de pruebas es **qué** es lo que hace el sistema. Por el contrario, las pruebas de caja blanca prueban el sistema pero atendiendo a **cómo** lo hace. En las pruebas de caja blanca, al contrario que en las de caja negra, sí son importantes los aspectos internos del software.

- **Prueba de estrés**. Este tipo de pruebas se realiza sobre el software ya acabado o en fase alfa. Es una de las típicas pruebas que se realiza una vez el software se acerca a su estado final y el objetivo del *testing* es generar una gran cantidad de datos para verificar si el software es robusto y no se ve afectado por la concurrencia. Muchas veces, cuando se desarrolla un software, este se prueba con una serie de casos de prueba a menudo muy sencillos y de forma aislada por el programador o analista. En este caso se ponen a trabajar con el sistema múltiples equipos como cliente y generalmente se utiliza software que imita el uso que le daría un usuario concreto (dando altas, bajas modificaciones, accediendo a datos, etc.). Esto provoca que el sistema se enfrente a una situación más real que hasta ahora. De esta prueba se tomarán tiempos y se podrá conocer cuál será el comportamiento del sistema una vez implantado.

- **Pruebas de integración**. Generalmente los proyectos están desarrollados por varios programadores o varios equipos de programación, los cuales trabajan sobre una parte del mismo. Existe un momento crítico y es cuando dichos programadores o grupos integran su trabajo para compilar un proyecto completo. Muchas veces, en ese momento surgen errores, incompatibilidades o fallos debidos a la integración de las distintas partes del software.

- **Pruebas de interfaces**. En algunos proyectos se desarrollan varios módulos de una aplicación concreta o varias aplicaciones que comparten información entre sí. Este tipo de pruebas trata de verificar que las aplicaciones o módulos son capaces de comunicarse entre sí de una manera eficiente y efectiva.

Una práctica popular y cada vez más habitual es la de distribuir de forma gratuita una versión no final del producto para que sean los propios consumidores los que la prueben. En ambos casos, a la versión del producto en pruebas anterior a la **versión final** (**máster**) se la denomina **versión beta;** y a dicha fase de pruebas, *beta testing*.

En ocasiones existe una versión anterior en el proceso de desarrollo, llamada **versión alpha**, en la que el programa aun estando incompleto presenta una funcionalidad básica y puede ser ya testeado.

Finalmente y antes de salir al mercado, es cada vez más habitual que se realice una fase llamada RTM *testing* (*release to market*), donde se comprueba cada funcionalidad del programa completo en entornos de producción.

HACIENDO HISTORIA...

Quizás uno de los fallos más históricos causados por un *bug* en sistemas de computación fue el que se produjo en enero del 2000 al sufrir las consecuencias del llamado efecto Y2K (*Y2K bug*).

8.2.2 EL PROCESO DE PRUEBAS

El proceso de pruebas consta de una serie de pasos que se van a abordar uno detrás de otro. Cada uno de estos pasos estará debidamente documentado. Pasemos a estudiar en profundidad cada uno de ellos.

8.2.2.1 Planificación de las pruebas: el plan de pruebas

En este primer paso se definen los objetivos de la prueba; en definitiva, **qué** se desea probar. Todo el mundo pensará que el objetivo será detectar errores, pero muchas veces eso no es lo más importante. En ocasiones se busca que el sistema ofrezca un rendimiento determinado, otras que la interfaz cumpla unas determinadas características, etc. En ocasiones, el software no presenta errores pero no cumple ciertos requisitos y no pasa las pruebas.

En este momento se debe decidir quién hace la prueba y bajo qué condiciones se va a realizar. Hay que tener en cuenta que los propios programadores no son los más adecuados para efectuar las pruebas, puesto que se limitarán a realizar las pruebas que ya han ido haciendo durante el desarrollo del software, y, por lo tanto, el objetivo de la prueba no tiene mucho sentido.

También hay que describir la estrategia que se va a seguir, es decir, cómo se va a realizar la prueba. Es importante en este momento decidir los recursos que se le van a asignar a las pruebas, incluyendo las personas, las máquinas donde se va a realizar, el tiempo que se va a dedicar, etc.

Hay que tener en cuenta que es imposible probar todo, la prueba exhaustiva no existe. Muchos errores del sistema saldrán en producción cuando el software ya está implantado, pero se intentará siempre que sea el mínimo número de ellos.

En esta fase se elaborará un plan de pruebas, que debería tener cubiertos, al menos, los siguientes apartados:

- **Introducción**. Breve introducción del sistema describiendo objetivos, estrategia, etc.
- **Módulos o partes del software que se van a probar**. Hay que detallar cada una de estas partes o módulos.
- **Características del software que se va a probar**. Características individuales o conjuntos de ellas.
- Características del software que no se van a probar.
- **Enfoque de las pruebas**. En el que se detallan, entre otros, las personas responsables, la planificación, la duración, etc.
- **Criterios de validez o invalidez del software**. En este apartado se registra cuándo el software puede darse como válido o como inválido especificando claramente los criterios.
- **Proceso de pruebas**. Se especificará el proceso y los procedimientos de las pruebas que hay que ejecutar.
- **Requerimientos del entorno**. Incluyendo niveles de seguridad, comunicaciones, necesidades hardware y software, herramientas, etc.
- **Homologación o aprobación del plan**. Este plan deberá estar firmado por los interesados o responsables del mismo.

Las demás fases del proceso de pruebas, como se puede entender, son el mero desarrollo del plan de pruebas anterior.

8.2.2.2 Preparación de los datos de prueba

En esta fase de las pruebas se diseñarán los casos de prueba. Los casos de prueba se diseñan para que haya una alta probabilidad de encontrar un fallo. Es importante en esta fase no ser redundante. Si se ha probado algo y funciona, no hace falta probarlo más.

Tenemos que tener en cuenta que la prueba no debe ser muy sencilla ni muy compleja. Si es muy sencilla, no va a aportar nada; y si es muy compleja, quizás sea difícil encontrar el origen de los errores.

RECUERDA

Probar es ejecutar casos de prueba uno a uno, pero el que un software pase todos los casos de prueba no quiere decir que el programa esté exento de fallos.

Como hemos visto, las pruebas solo encuentran o tratan de encontrar aquellos errores que van buscando, luego es muy importante realizar un buen diseño de las pruebas con buenos casos de prueba puesto que se aumenta de esta manera la probabilidad de encontrar fallos.

RECUERDA

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y salidas esperadas diseñadas para un objetivo concreto.

8.2.2.3 Codificación de las pruebas

Una vez diseñados los casos de prueba hay que generar las condiciones necesarias para poder ejecutar dichos casos de prueba. Habrá que codificarlos en muchos casos generando ***sets* o conjuntos de datos**. En estos *sets* de datos hay que incluir tanto datos válidos como datos inválidos o incluso algunos datos fuera de rango o disparatados.

También habrá que preparar las máquinas sobre las que se van a hacer las pruebas: instalar el software necesario, los usuarios de sistema, realizar carga del sistema, etc.

8.2.2.4 Ejecución de las pruebas

Una vez definidos los casos de prueba y establecido el entorno de las pruebas, es el momento de la ejecución de las mismas.

Se irán ejecutando los casos de prueba uno a uno, y, en el momento que se detecte algún error, lo que hay que hacer es aislarlo y anotar la acción que se estaba probando, el caso, el módulo, la fecha, la hora, los datos utilizados, etc. De esa manera se intentará documentar lo más detalladamente posible el error.

En caso de que se produzcan errores aleatorios, también hay que registrarlos anotando este hecho.

RECUERDA

Casi siempre que se encuentra un error en un módulo de software, si se insiste, se encontrarán más.

8.3 DOCUMENTACIÓN

La documentación es todo aquel conjunto de manuales impresos o en formato digital y cualquier otra información descriptiva que explique una aplicación informática o programa.

8.3.1 ESTRUCTURA Y TIPOS DE DOCUMENTACIÓN

Toda aplicación se puede contemplar desde dos aspectos: su **descripción física** o **técnica** (cómo es físicamente, analizando los componentes que la constituyen tales como diagrama de clases, ficheros que componen el sistema, interfaces, descripción a fondo de cada una de las clases, descripción del sistema de almacenamiento en bases de datos o ficheros, etc.), así como los elementos de interconexión o interfaces con otros sistemas; y su **descripción funcional** (funcionamiento del sistema, funciones de cada uno de sus componentes, cómo interactúan unos con otros, reglas o normas de comunicación, etc.). La documentación, como ya veremos más adelante, debe cubrir como mínimo ambas facetas, la faceta **técnica** (información para los informáticos) y la **funcional** (información para todos, especialmente para los usuarios).

La documentación es un proceso que comienza desde el principio del proyecto y es algo que nunca termina. Los proyectos, según el **ciclo de vida clásico**, van pasando por las siguientes etapas:

- **Fase inicial**. En esta fase se planifica el proyecto, se hacen estimaciones, se conviene si el proyecto es rentable o no, etc. Es decir, se establecen las bases de cómo se van a desarrollar el resto de fases del proyecto. En un símil con la construcción de un edificio sería el ver si se dispone de licencia de construcción, cuánto me va a costar el edificio, cuántos trabajadores voy a necesitar para construirlo, quién lo va a construir, etc. La fase de planificación y estimación es la más complejas de un proyecto. Para esto se necesita gente con experiencia tanto en la elaboración de proyectos como en las plataformas en las que se va a realizar el mismo. De esta fase surgen muchos documentos tanto de planificación (general y detallada) como de estimaciones, en los que se incluyen datos económicos, posibles soluciones al problema y sus costes, etc. Son documentos que se realizan a alto nivel y se acuerdan con la dirección de la empresa o con las personas responsables del proyecto. En estas fases iniciales se suelen tomar decisiones que a veces afectan a todas las demás fases del proyecto, con lo cual tiene que estar todo bien documentado, detallado y soportado por datos concretos.

- **Análisis**. En esta fase se analiza el problema. Consiste en recopilar, examinar y formular los requisitos del cliente y analizar cualquier restricción que se pueda aplicar. Por lo tanto, todas las entrevistas con el cliente tienen que estar registradas en documentos y generalmente esos documentos se consensuan con el cliente; y, desde mi punto de vista, algunos tienen hasta caracter contractual. Yo, como jefe de desarrollo, en algunos proyectos he hecho firmar al cliente un documento de requisitos de la aplicación en el cual mi equipo se compromete a realizar las especificaciones indicadas por el cliente y también el cliente se compromete a no variar sus necesidades hasta por lo menos terminar una primera *release*. Como puedes ver, es un documento que obliga a ambas partes a cumplir con lo acordado y en muchas ocasiones establece un marco de relación entre cliente y desarrollador.

- **Diseño**. Esta fase consiste en determinar los requisitos generales de la arquitectura de la aplicación y en dar una definición precisa de cada subconjunto de la aplicación. En esta fase los documentos ya son más técnicos. Se suelen crear dos documentos de diseño, uno genérico en el que se tiene una visión de la aplicación más general, y otro detallado, en el que se profundizará en los detalles técnicos de cada módulo concreto del sistema. Estos documentos los realizarán los analistas con la supervisión del jefe de proyecto.

- **Codificación o implementación**. Esta fase consiste en la implementación del software en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño. Durante esta fase se crea documentación muy detallada en la que se incluye código. Aunque mucho código se suele comentar en el mismo programa, también se tienen que generar documentos donde se indica, por ejemplo, para cada función las entradas, salidas, parámetros, propósito, módulos o librerías donde se encuentra, quién la ha creado, cuándo, las revisiones que se han realizado de la misma, etc. Como puedes ver, el detalle es máximo teniendo en cuenta que ese código en un futuro va a tener que ser mantenido por la misma o seguramente otra persona, y, a veces, toda la información que pueda recibir es poca.

- **Pruebas**. En esta fase se llevarán a cabo pruebas para garantizar que la aplicación se programó de acuerdo con las especificaciones originales y que los distintos programas de los que consta la aplicación están perfectamente integrados y preparados para la explotación. Como se ha visto anteriormente, las pruebas son de todo tipo. Yo clasificaría la documentación de las pruebas en dos bloques diferentes. Existen unas **pruebas funcionales** en las que se prueba que la aplicación hace lo que tiene que hacer con las funciones acordes a los documentos de especificaciones que se establecieron con el cliente, y esas pruebas se deberían realizar con el cliente delante. Mientras se están realizando las pruebas, se tienen que hacer todo tipo de anotaciones para luego plasmarlas en un documento al que tendrá que darle el visto bueno el cliente (que por ese motivo estuvo en las pruebas).

En ese documento se van detallando fallos tanto de la propia aplicación como modificaciones a la misma si no cumple con las especificaciones iniciales. En caso de que haya discrepancia, se suele consultar documentación anterior para que no se incluyan en este punto funcionalidades nuevas o variaciones de las funcionalidades. En otro documento aparte se detallarán los resultados de las **pruebas técnicas** realizadas. A estas pruebas ya no hace falta que asista el usuario o cliente puesto que son de carácter meramente técnico. En estas pruebas se harán cargas reales, se someterá la aplicación y el sistema a estrés, etc.

- **Explotación**. En esta fase se instala el software en el entorno real de uso y se trabaja con él de forma cotidiana. Generalmente es la fase más larga y suelen surgir multitud de incidencias, nuevas necesidades, etc. Toda esta información se suele detallar en un documento en el que se recogen los errores o fallos detectados, se intentará que sea lo más explícito posible puesto que luego los programadores y analistas deben revisar estos fallos o *bugs* y darles la mejor solución posible. También surgirán otras necesidades que se van a ir detallando en un documento y que pasarán a realizarse en operaciones de mantenimiento.

- **Mantenimiento**. En esta fase se realizan todo tipo de procedimientos correctivos (corrección de fallos) y actualizaciones secundarias del software (mantenimiento continuo) que consistirán en adaptar y evolucionar las aplicaciones. Para realizar las labores de mantenimiento hay que tener siempre delante la documentación técnica de la aplicación. Sin una buena documentación de la aplicación, las labores de mantenimiento son muy difíciles y su garantía en ese caso sería escasa. Todas las operaciones de mantenimiento tienen que estar documentadas porque se tiene que conocer quién ha realizado la operación, qué ha hecho y cómo. También tienen que estar documentadas porque deberían probarse por otra persona distinta al programador.

En cada una de estas fases se generan uno o más documentos. En ningún proyecto es viable comenzar la codificación sin haber realizado las fases anteriores porque eso equivaldría a un desastre absoluto. Además, la documentación debe ser útil y estar adaptada a los potenciales usuarios de dicha documentación (cuando se crea un coche existen los manuales de usuario y los manuales técnicos para los mecánicos. Para qué quiero yo como usuario saber dónde están situados los inyectores, las bujías o la trócola si nunca la voy a cambiar. A mí lo que me interesa es saber cómo se regula el volante, cómo funciona la radio, etc.).

Visto esto, debo decir que en cualquier aplicación, **como mínimo**, se deberán generar los siguientes documentos:

- **Manual de usuario**. Es, como ya se comentó anteriormente, el manual que utilizará el usuario para desenvolverse con el programa. Deberá ser autoexplicativo y de ayuda para el usuario. Este manual debe servirle al usuario para aprender cómo se maneja la aplicación y qué es lo que hay que hacer y lo que no. Si como técnico no vas a hacer un manual que le sirva al usuario en su comienzo o práctica diaria, es mejor no hacerlo o realizar otro tipo de documentación.

- **Manual técnico**. Es el manual dirigido a los técnicos (el manual para los mecánicos citado anteriormente). Con esta documentación, cualquier técnico que conozca el lenguaje con el que la aplicación ha sido creada debería de poder conocerla casi tan bien como el personal que la creó.

- **Manual de instalación**. En este manual se explican paso a paso los requisitos y cómo se instala y pone en funcionamiento la aplicación.

Desde la experiencia, hay que recalcar una vez más la importancia de la documentación, puesto que sin documentación una aplicación o programa es como un coche sin piezas de repuesto, cuando tenga un problema o haya que repararlo no se podrá hacer nada.

8.3.2 GENERACIÓN AUTOMÁTICA DE DOCUMENTACIÓN

Las aplicaciones o programas tienen que estar perfectamente documentados, de lo contrario sería muy difícil mantener el código. En Java, la documentación del código se escribe dentro del mismo, lo cual es verdaderamente útil. Java, además, tiene una herramienta, llamada Javadoc, que extrae los textos y comentarios del código fuente y los transforma en páginas web (formato HTML).

Para escribir comentarios que Javadoc interprete, hay que insertarlos entre los caracteres /** y */. Dentro de estos caracteres especiales ya podemos escribir en HTML utilizando sus etiquetas y utilizando también otras etiquetas específicas como:

- **@author**. Identifica al autor del código.
- **@version**. Identifica la versión del código.
- **@since**. Especifica la fecha de creación del código.
- **@param <nombre>**. Describe el parámetro "nombre". Una línea por cada parámetro.
- **@return**. Especifica el valor de retorno del método. Si la función devuelve void no hay que especificarlo.
- **@deprecated**. Si la clase, método o ítem documentado está obsoleto.
- **@see**. Utilizado si se quiere dirigir al lector a otro apartado.
- **@link**. Utilizado si se quiere dirigir al lector al recurso especificado por la URL.

Veamos un ejemplo de documentación para el programa de colas visto en el tercer capítulo. Tenemos el código que se detalla a continuación:

```
/**
 * clase cola
 * @author Juan Carlos
 * @version 1.0
 * @since 21/04/2014
 * <br>
 * <p>Esta clase corresponde a un ejemplo de implementaci&oacute;n de una estructura
 * din&aacute;mica como es una cola.</p>
 */
public class Cola {
```

RECUERDA

Para generar la documentación de la clase anterior se ha ejecutado desde la línea de comandos lo siguiente:

```
javadoc Cola.java
```

Tras ejecutar Javadoc, el cual se puede ejecutar desde el propio IDE (NetBeans o Eclipse) o desde línea de comandos (si utilizas Geany deberás ejecutarlo de esta manera), obtenemos una serie de páginas web con la documentación de la aplicación.

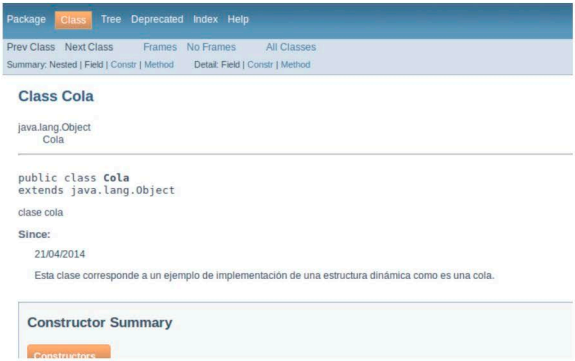


Figura 8.1. *Detalle de la generación HTML de Javadoc*

HERRAMIENTAS PARA GENERACIÓN DE AYUDAS: JAVAHELP

Para que una aplicación tenga un menú de ayuda integrado en la propia aplicación generalmente se utiliza **JavaHelp**. JavaHelp es una extensión de Java que va a permitir implementar ventanas de ayuda dentro de Java. Estas ventanas de ayuda se basan en ficheros XML y HTML, que son los que se mostrarán al usuario.

8.4 PROCESO DE INGENIERÍA DEL SOFTWARE

Por software entendemos el equipamiento o soporte lógico de un sistema informático. Lo constituyen el conjunto de componentes lógicos —y, por tanto, no tangibles y no físicos— necesarios para llevar a cabo una tarea específica en nuestro sistema.

Es un componente imprescindible en todo sistema informático, que comunicará y dará órdenes al hardware para que se lleven a cabo todas las tareas que se el usuario del sistema le encomiende.

Podemos definir el software como el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación (definición extraída del estándar 729 de IEEE).

El concepto ya fue empleado por **Charles Babbage** como parte de su máquina diferencial, en forma de diferentes secuencias de instrucciones leídas desde memoria. Posteriormente, **Alan Turing**, con su máquina de Turing y su teoría de la computación, desarrolló la teoría que forma la base del software moderno.

ALAN TURING

Alan Turing (1.912-1.954) fue un matemático, informático teórico y criptógrafo inglés considerado uno de los padres de la Ciencia de la Computación siendo el primer antecedente de la informática moderna. Formuló la Teoría de la Computación, hoy día ampliamente aceptada, y contribuyó a combatir a los alemanes en la Segunda Guerra Mundial ayudando a descifrar su potente máquina Enigma.
Su emergente carrera se cortó bruscamente cuando fue acusado y procesado por ser homosexual siendo castrado químicamente y suicidándose al poco tiempo.

Figura 8.2. *Ficha personal de Alan Turing*

El software es un elemento con unas características muy particulares que lo llevan a que diferentes acciones sobre el mismo, como el desarrollo o el mantenimiento, sean también muy particulares. Estas características son:

- El software es lógico, no físico.
- El software se desarrolla, no se fabrica.
- El software no se estropea.
- En ocasiones se puede construir a medida.

¿SABÍAS QUE?

La palabra "software" como tal fue empleada por primera vez en 1957 por J.W. Tukey (1915-2000), quien, en un artículo de 1958 en la publicación *American Mathematical Monthly*, empleó por primera vez el término "*computer software*" en un contexto computacional donde hablaba de la necesidad de aprovechar las capacidades de cálculo de las computadoras para permitir a los programadores escribir y organizar complejos conjuntos de instrucciones que luego se traducirán a un lenguaje comprensible por las máquinas para que puedan ser ejecutadas.
J.W. Tukey también acuñó otro término imprescindible en la tecnología computacional, la palabra "bit" como contracción de "digito binario", por sus siglas en inglés (*binary digit*).

8.4.1 FASES DEL PROCESO DE INGENIERÍA SOFTWARE

Todo software en su creación y desarrollo pasa por una serie de etapas conocidas como las "fases del **ciclo de vida del software**" dentro de la disciplina dedicada a esta cuestión: la ingeniería del software.

El objetivo de la ingeniería del software es proporcionar un marco de trabajo para construir software con mayor calidad.

El término "ciclo de vida del software" describe el desarrollo de software, desde la fase inicial hasta la fase final. El propósito de este modelo es definir las distintas fases intermedias que se requieren para la validación del desarrollo de la aplicación (garantizar que el software cumpla los requisitos para la aplicación) y la verificación de los procedimientos de desarrollo (asegurar que los métodos utilizados son apropiados).

Este tipo de modelos se desarrollan tomando como punto de partida que es muy costoso rectificar los errores que se detectan tarde dentro de la fase de implementación. El ciclo de vida permite que los errores se detecten lo antes posible, y, por lo tanto, permite a los desarrolladores concentrarse en la calidad del software, en los plazos de implementación y en los costos asociados.

Figura 8.3. *Ciclo de vida del software*

En general, todo ciclo de vida de un software consta de las siguientes etapas:

- **Definición de necesidades**. Consiste en realizar una primera aproximación al proyecto y definir a grandes rasgos las necesidades
- **Análisis**. Consiste en recopilar, examinar y formular los requisitos del cliente y examinar cualquier restricción que se pueda aplicar. En este paso se realiza un análisis en profundidad de la arquitectura hardware y software del sistema.
- **Diseño**. Se determinarán los requisitos generales de la arquitectura de la aplicación y se dará una definición precisa de cada subconjunto de la aplicación.
- **Codificación** (programación e implementación). Consiste en la implementación del software en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño.

- **Pruebas**. Se llevará a cabo una prueba individual de cada subconjunto de la aplicación para garantizar que se implementaron de acuerdo con las especificaciones y se garantizará que los diferentes módulos se integren con la aplicación.
- **Validación**. Se garantizará que el software cumple con las especificaciones originales y se instalará el software en el entorno real de uso.
- **Mantenimiento y evolución**. Se realizarán los procedimientos correctivos (mantenimiento correctivo) y las actualizaciones secundarias del software (mantenimiento continuo).

El orden y la presencia de cada uno de estos procedimientos en el ciclo de vida de una aplicación dependen del tipo de modelo de ciclo de vida acordado entre el cliente y el equipo de desarrolladores.

8.4.2 MODELOS DEL PROCESO DE INGENIERÍA

Como se ha dicho antes, dependiendo del modelo de ciclo de vida, los técnicos actuarán de una manera o de otra. Lo importante es que el producto final se ajuste a las especificaciones del cliente y el tiempo de desarrollo y el coste del mismo sean lo más reducidos posible.

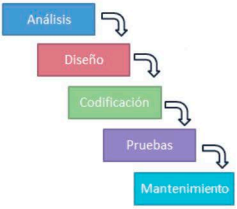


Figura 8.4. *Modelo de desarrollo software clásico o en cascada*

El ciclo de vida clásico siempre ha sido por fases, se acometía una fase detrás de otra y por esa razón se denominaba "en **cascada**". Actualmente se sigue otro tipo de paradigmas, dado que los lenguajes de programación y las herramientas existentes permiten realizar el desarrollo de otra manera más eficiente.

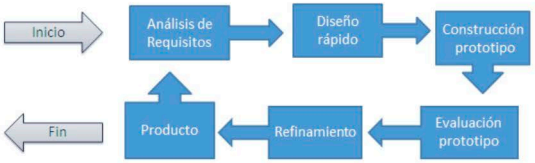


Figura 8.5. *Modelo de desarrollo por prototipado o software de prototipos*

Un modelo muy utilizado es el **prototipado**. El objetivo es crear una versión del producto que se irá refinando cada vez más hasta obtener el modelo definitivo. Este tipo de paradigma se utiliza cuando no se sabe de forma clara cómo tiene que ser el producto final.

Actualmente en los modelos orientados a objetos se utilizan otros paradigmas más laxos. Debido a la naturaleza del desarrollo de la programación orientada a objetos se ve necesario utilizar modelos iterativos como el prototipado pero eliminando las fronteras entre las distintas fases para que la forma de trabajo sea más dinámica.

8.4.3 REQUISITOS: CONCEPTO, EVOLUCIÓN Y TRAZABILIDAD

El **análisis de requisitos** es una de las primeras fases en las que se sustenta el ciclo de vida del software. Las principales técnicas que se suelen utilizar son las entrevistas. Esta primera fase no es un trabajo fácil. Hay que tener habilidades sociales para tratar con los clientes y tener experiencia en el desarrollo de software para poder abordar esta fase con éxito.

Una de las principales actividades es la identificación de actores. Hay que conocer las necesidades de todas y cada una de las personas que van a utilizar el sistema, y una vez tengamos claro los intervinientes, hay que realizar entrevistas o talleres con grupos de personas para terminar gestando un prototipo o una serie de casos de uso.

LOS CASOS DE USO

Los casos de uso son una herramienta eficaz en el análisis de requisitos puesto que se trata el sistema como una entidad de caja negra y se analiza la interacción con las entidades externas (usuarios y otros intervinientes). De esa forma, el analista puede comprender los *inputs* que recibe el sistema y la salida que de él se espera.

Aunque las entrevistas han sido la herramienta clásica más utilizada, también existe otro tipo de herramientas como los talleres o reuniones grupales, en los que pueden salir muchos de los requisitos de un sistema como los *brainstormings* o las JAD (*Joint Application Development* - diseño conjunto de la aplicación). *Brainstorming* y JAD no se parecen en nada. En los *brainstormings* se reúnen varias personas que van lanzando ideas una detrás de otra con el fin de tomar nota de cada una de ellas y luego tenerlas en cuenta para hacer un guión o documento en el que se le dé cuerpo a dichas ideas. En las JAD el proceso es más ordenado y dirigido: se sigue un guión con una serie de puntos que hay que tratar, y cada uno de los puntos se trata y analiza con la parte implicada. El inconveniente de las JAD es que el proceso de análisis se demora más en el tiempo puesto que hay que concertar citas, previamente hay que prepararlas y en ocasiones hay que preparar citas a las que tienen que asistir varios participantes, lo que hace que el proceso se pueda alargar por cuestiones de agenda.

A la finalización del proceso de análisis de requisitos se debería firmar un contrato en el que las partes firmantes (usuarios y desarrolladores) acuerden la realización de un software con unos requisitos determinados. En ocasiones no hace falta firmar un contrato, sino los documentos que se hayan ido gestando durante esta fase.

8.4.4 HERRAMIENTAS CASE: HERRAMIENTAS DE INGENIERÍA SOFTWARE, ENTORNOS DE DESARROLLO, HERRAMIENTAS DE PRUEBA, DE GESTIÓN DE LA CONFIGURACIÓN, HERRAMIENTAS PARA MÉTRICAS

Las **herramientas CASE** (*Computer Aided System dEvelopment*) nacieron en los años 70 y su objetivo era aumentar la productividad y el control en el diseño y desarrollo del software, haciendo el proceso algo más calculado y protocolarizado.

El objetivo de estas herramientas era ayudar a los integrantes del equipo de desarrollo a generar software de una manera más reglada y para ello comenzaron a aparecer numerosas aplicaciones que cubrían ciertos aspectos del desarrollo del software, como métricas, análisis, estimaciones, etc.

IBM —que era la empresa más importante en el mundo de la informática en su momento— con el boom de los grandes *mainframes* generó herramientas que abarcaban toda la vida del software, pero en la actualidad se ha visto que este tipo de herramientas no son operativas y lo que hoy existe son herramientas muy efectivas y precisas en ciertas partes del ciclo de vida.

Surgieron así las:

- **Upper CASE**. Cubrían las primeras fases del ciclo de vida del software.
- **Lower CASE**. Cubrían las últimas fases del ciclo de vida del software.
- **I-CASE** (*Integrative* CASE). Cubrían todas las fases del ciclo de vida del software.

Las ventajas que proporciona el desarrollo con herramientas CASE, y la razón por la cual las empresas invirtieron miles de millones en ellas, son las siguientes:

- Mejora del rendimiento.
- Mejora la documentación y comunicación entre el equipo de desarrollo.
- Las aplicaciones generadas son más fiables.
- El trabajo de diseño es menos costoso.
- Se abarata el coste de desarrollo.
- Se generan menos errores durante el ciclo de vida del software.
- El mantenimiento del software es más sencillo y barato.

Algunas herramientas CASE están orientadas al:

- **Modelado de datos**. Son herramientas que permiten definir los procesos de negocio del sistema. Estas herramientas se utilizan en las primeras fases del desarrollo de software.
- **Herramientas de generación de código**. Estas herramientas aparecieron en un momento en el que había mucha necesidad de crear abundante software, y en poco tiempo. Además, estas herramientas tenían la virtud de estar exentas de errores. Se pensaba que en el futuro estas herramientas reemplazarían a los programadores pero al final se ha visto que no ha sido del todo así. Actualmente, en vez de herramientas de generación de código se utilizan muchas librerías y *frameworks*, lo que hace que el desarrollo de programas sea mucho más rápido y eficiente.
- **UML** (*Unified Modelling Language*). UML combina métodos de modelado y diagramas que pueden ser utilizados durante todo el ciclo de vida del software. Se utiliza sobre todo para desarrollar sistemas basados en programación orientada a objetos.
- **Documentación**. Estas herramientas sirven para aliviar a los desarrolladores del aburrimiento que supone tener que crear la documentación del sistema. Es la tarea menos creativa para el desarrollador, pero una de las más importantes. Estas herramientas estandarizan la documentación, mejoran la apariencia, y permiten su modificación y actualización de forma colaborativa.

8.5 TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:
a) La palabra "software", como tal, fue empleada por primera vez en 1957 por J.W. Tukey.
b) Los años noventa fueron escenario de una gran crisis del software, provocada por su falta de calidad.
c) Las pruebas de caja blanca prueban el sistema, pero atendiendo a cómo funciona internamente.
d) En la prueba de estabilidad se somete la aplicación a una carga que va cambiando y generando picos de demanda.

2 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Las pruebas de estrés se realizan sobre el software ya acabado o en fase alfa.
b) El software es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación.
c) La documentación es un proceso que comienza desde el principio del proyecto.
d) La prueba de validación consiste en demostrar que un programa cumple con sus especificaciones.

3 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Generalmente, para evaluar los criterios de calidad se realizan RTF o revisiones técnicas formales.
b) En la versión alpha, el programa, aun estando incompleto, presenta una funcionalidad básica y puede ser ya testeado.
c) En la fase llamada RTM *testing* se comprueba cada funcionalidad del programa completo en entornos de producción.
d) El concepto de software fue empleado por Alan Turing en su máquina diferencial.

4 ¿Cuál de las siguientes afirmaciones es falsa?:
a) Un caso de prueba es un conjunto de entradas, condiciones de ejecución y salidas esperadas diseñadas para un objetivo concreto.
b) El software siempre se construye a medida.
c) Las pruebas de rendimiento sirven para determinar lo rápido que realiza una tarea un software en condiciones particulares de trabajo.
d) La documentación es el conjunto de manuales impresos o en formato digital y cualquier otra información descriptiva que explique una aplicación informática o programa.

5 ¿Cuál de las siguientes afirmaciones es falsa?:
a) El ciclo de vida clásico se denominaba "en cascada".
b) En las pruebas de aplicaciones base se usan llamadas al sistema durante actividades específicas de una aplicación, como uso de disco o llamadas a rutinas de gráficos, ejecutándolas aisladamente.
c) Quizás uno de los fallos más conocidos causados por un *bug* en sistemas de computación fue el que se pudo producir en enero del 2000.
d) Durante las pruebas se pretende detectar errores de programación o *bugs*.

Solucionario de los test de conocimientos

- CAPÍTULO 1: 1B, 2B, 3B, 4C, 5C, 6D
- CAPÍTULO 2: 1A, 2A, 3A, 4C, 5B
- CAPÍTULO 3: 1C, 2B, 3C
- CAPÍTULO 4: 1B, 2B, 3C, 4C, 5A
- CAPÍTULO 5: 1B, 2B, 3C, 4C, 5A
- CAPÍTULO 6: 1C, 2C, 3C, 4B, 5B
- CAPÍTULO 7: 1D, 2C, 3B, 4A
- CAPÍTULO 8: 1D, 2D, 3D, 4B, 5B

Índice alfabético

CERTIFICADO DE PROFESIONALIDAD

Programación Orientada a Objetos

La presente obra está dirigida a los estudiantes del certificado de profesionalidad **Programación con Lenguajes Orientados a Objetos y Bases de Datos Relacionales**, en concreto a los del módulo formativo **Programación Orientada a Objetos**, y a toda aquella persona que quiera aprender la programación orientada a objetos con Java.

Los contenidos incluidos en este libro abarcan temas muy interesantes, como la programación orientada a objetos, la programación web y el acceso a bases de datos relacionales, así como conceptos de ingeniería del software.

Los capítulos incluyen notas, esquemas, ejemplos y test de conocimientos con el propósito de facilitar la asimilación de las cuestiones tratadas. Al terminar de comprender y asimilar esta obra, el lector estará capacitado para empezar a desarrollar programas orientados a objetos en Java, que, en la actualidad, es uno de los lenguajes con más futuro.

9 788499 645094

ra-ma.es

Ra-Ma®