CF GRADO SUPERIOR

CICLOS FORMATIVOS

R.D. 1538/2006

Programación

Ra-Ma®

www.ra-ma.es/cf

JUAN CARLOS MORENO

Programación

JUAN CARLOS MORENO PÉREZ

Ra-Ma®

La ley prohíbe
fotocopiar este libro

PROGRAMACIÓN

Editado por:
RA-MA Editorial
Calle Jarama, 3A, 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**
ISBN: 978-84-9964-088-4
Depósito Legal: M-24.444-2011
Maquetación: Gustavo San Román Borrueco
Diseño de Portada: Antonio García Tomé
Filmación e Impresión: Closas-Orcoyen, S. L.
Impreso en España

Índice

Introducción

Este libro tiene como objetivo el servir de referencia al alumno en el módulo de programación. Este módulo es de los más prácticos del ciclo y por lo tanto he procurado hacer un libro en el que se complemente la parte práctica con los contenidos teóricos. Hay que tener en cuenta que este libro debe de servir para formar a profesionales, por lo tanto debe de ser lo más útil posible para el alumno y debe proporcionar conocimientos prácticos y actualizados.

La estructura del libro es lo más didáctica posible. He intentado que los conceptos en el libro sean fáciles de comprender acompañándolos de muchos ejemplos, consejos, notas, aclaraciones, etc.

Como profesor, alumno, programador, consumidor de libros, tutoriales, etc. He intentado seguir los siguientes principios a la hora de escribir el libro:

- Mirando se aprende.
- No escribas para otros lo que a ti te costaría entender. Parece absurdo pero a veces ocurre.
- Los ejemplos tienen que ir al grano y ser cortos. Los ejemplos largos aburren.
- No compliques los ejemplos, ya habrá otro que lo tome y lo complique.

En realidad estos principios son simplemente lo que yo desearía que fuera el libro si tuviese que trabajar con él. Espero que tu y yo coincidamos en esto.

En el libro se ha optado por utilizar Java como lenguaje para desarrollar todos los conceptos del decreto. Se podría haber utilizado C++ como lenguaje base, pero dada la proyección de Java no me parecía lo más útil a largo plazo. Tampoco es operativo utilizar los dos lenguajes porque como dicen los abuelos: "El que mucho abarca poco aprieta". Alguna de las razones para utilizar Java son la independencia de la plataforma, la gran variedad de herramientas de desarrollo, la modularidad y reutilización, la gran cantidad de recursos disponibles, la facilidad para programar en Internet con este lenguaje, etc.

El alumno además de manejar el libro y el material adicional, deberá de investigar, 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.

En el material adicional vas a encontrar todo el software utilizado y necesario para practicar con este libro y mucha más información práctica. Te recomendamos que lo examines para conocer la información que contiene. Verás las intrucciones de descarga en el apartado "Material adicional".

Los ejercicios propuestos con un nivel de dificultad alto se han especificado al comienzo de su exposición indicando entre paréntesis el texto "Ejercicio de dificultad alta". Se han catalogado de esa forma aquellos ejercicios que son más complicados o que requieren de un mayor tiempo de programación que los demás del capítulo. Obviamente este nivel de dificultad es relativo al nivel exigido en cada capítulo.

1 Elementos de un programa informático

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer qué es un programa, un lenguaje de programación y las diferencias entre lenguajes de programación como Java y C o C++.
- ✓ Reconocer el aspecto de un programa básico en Java y sus características principales.
- ✓ Instalar y utilizar un IDE.
- ✓ Compilar y ejecutar programas sencillos en Java dentro y fuera de un Entorno de desarrollo.
- ✓ Conocer y utilizar fundamentos básicos del lenguaje Java como los tipos de datos, constantes, literales, variables, comentarios, operadores y expresiones.
- ✓ Identificar las ventajas y limitaciones de Java frente a otros lenguajes de programación.

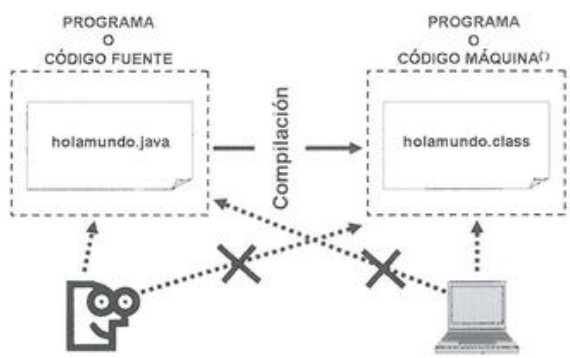
La información de este capítulo muchas veces es un resumen y en ocasiones no trata en profundidad ciertos aspectos. No obstante, el alumno en la sección de bibliografía puede encontrar libros y páginas aconsejadas en los que puede ampliar o contrastar la información en este libro proporcionada.

1.1 PROGRAMA Y LENGUAJES DE PROGRAMACIÓN

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

Todo el mundo 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 click sobre él e incluso algunos usuarios más avanzados ejecutan comandos desde un intérprete de comandos o consola. Si una vez has tenido la curiosidad de abrir un programa con un bloc de notas o editor de texto te habrás dado cuenta 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:



(*) En Java es bytecode. Interpretable por la máquina virtual de Java.

Figura 1.1. Programas en código fuente y máquina

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 éste 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 en 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. 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).

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.

1.1.1 EL LENGUAJE JAVA

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 radica 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.

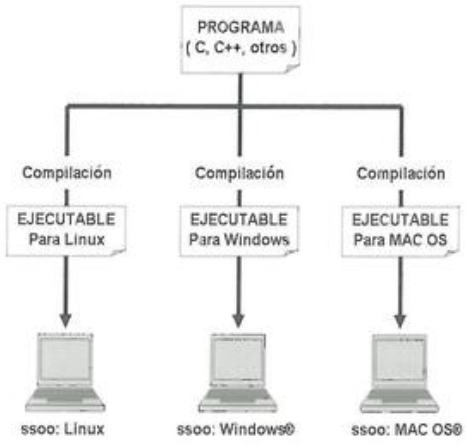


Figura 1.2. Recompilación del programa para cada sistema operativo

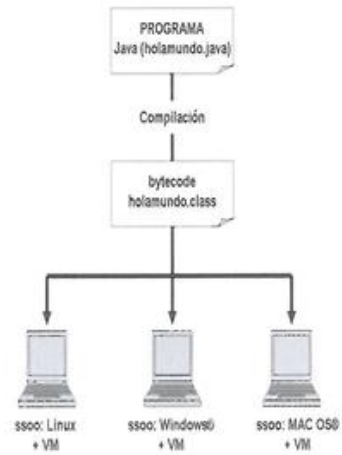


Figura 1.3. En Java una única compilación

Como puede apreciarse en las figuras anteriores, en Java, una vez compilado el programa, se puede ejecutar en cualquier plataforma solamente con tener instalada la máquina virtual (*Virtual Machine* – VM) de Java. Sin embargo en C, C++ u otro lenguaje, deberemos recompilar el programa para el sistema destino con la consiguiente pérdida de flexibilidad.

Por lo tanto, Java es un compilador y a la vez un intérprete. El compilador compila a bytecode y el intérprete se encargará de ejecutar ese código intermedio en la máquina real.

Recuerda
Java es multiplataforma y programas en Java pueden ser ejecutados en Windows®, GNU/Linux y Mac OS X entre otros sistemas.

James Gosling trabajaba para Sun Microsystems® y fue el diseñador de Java en 1990. El primer nombre que tuvo Java fue OAK y tuvo como referentes C y C++ (de hecho se parece mucho a ellos en el aspecto, pero la filosofía de funcionamiento es totalmente distinta). SUN® desarrollo este lenguaje en principio con otra orientación, la idea es que fuese utilizado en microelectrónica y sistemas embebidos. Lo que nunca se pensó SUN® es la repercusión y evolución que tendría más tarde este lenguaje.

Cuatro razones para aprender Java
1. Por el futuro y presente que tiene.
2. Es un lenguaje sencillo.
3. Es un lenguaje orientado a objetos.
4. Es independiente de la plataforma.

1.1.2 EL JDK

El JDK (*Java Development Kit*), aunque no contiene ninguna herramienta gráfica para el desarrollo de programas, sí que contiene aplicaciones de consola y herramientas de compilación, documentación y depuración. El JDK incluye el JRE (*Java Runtime Environment*) que consta de los mínimos componentes necesarios para ejecutar una aplicación Java, como son la máquina virtual y las librerías de clases.

El JDK contiene, entre otras, las siguientes herramientas de consola:

- **java**. Es la máquina virtual de Java.
- **javac**. Es el compilador de Java. Con él es posible compilar las clases que desarrollemos.
- **javap**. Es un desensamblador de clases.
- **jdb**. El depurador de consola de Java
- **javadoc**. Es el generador de documentación.
- **appletviewer**. Visor de *Applets*.

Importante
Una vez descargado e instalado el JDK hay que modificar los valores de dos variables de entorno:
- Variable PATH. Apunta donde está situado el directorio bin del JDK.
- Variable CLASSPATH. Apunta donde están situadas las clases del JDK.

Podemos descargar y utilizar varios JDK simplemente modificando los valores de ambas variables.

A FONDO

CONOCIENDO LA VERSIÓN DE JAVA

Para conocer la versión de java con la que estamos trabajando basta con ejecutar lo siguiente en una shell o intérprete de comandos:

```
java -version
```

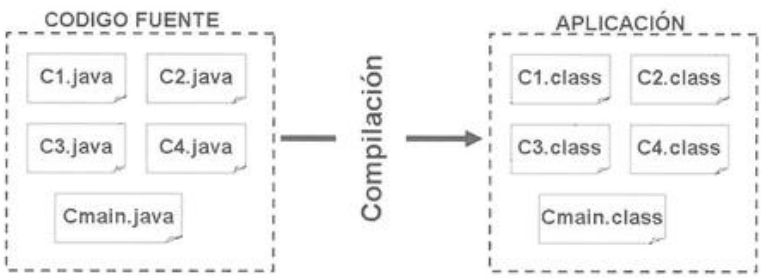
Y aparecerá en la ventana algo parecido a esto:

```
C:\Documents and Settings\JUAN CARLOS>java -version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)
```

1.1.3 LOS PROGRAMAS EN JAVA

Los programas o aplicaciones en Java se componen de una serie de 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 principal o *main()* situada en una clase. A partir de aquí se van creando objetos a partir de las clases y se va ejecutando la aplicación. El *main()* es un método estático (ya se explicará esto más adelante) el cual puede empezar a crear los objetos, incluidos los de su propia clase.



EJECUCIÓN

java Cmain.class

Figura 1.4. Proceso de compilación de un programa

1.2 ESTRUCTURA Y BLOQUES FUNDAMENTALES DE UN PROGRAMA

En este apartado se va a ver el programa de inicio por excelencia en cualquier lenguaje de programación y se comentará cada una de sus líneas. El proceso de compilación y ejecución se explica en el siguiente apartado.

```
public class holamundo {
/* programa holamundo*/
    public static void main(String[] args) {
          /* lo único que hace este programa es mostrar
            la cadena "Hola Mundo" por pantalla*/
          System.out.println("Hola Mundo");
    }
}
```

Los comentarios

Existen comentarios de una línea solamente (//) y comentarios multilínea (/* */).

- //. Estos comentarios comienzan en la doble barra y terminan hasta el final de la línea.
- /* */ . Estos comentarios comienzan con los caracteres /* y terminan con los caracteres */ y se pueden extender múltiples líneas.

- La clase *holamundo*

En java generalmente cada clase es un fichero distinto. Si existieran varias clases en el fichero, la clase cuyo nombre coincide con el nombre del fichero debería de llevar el modificador public (public class holamundo) y es la que se puede utilizar desde fuera del fichero. Las clases tienen el mismo nombre que su fichero .java y es importante que mayúsculas y minúsculas coincidan. La clase abarca desde la primera llave que abre hasta la última que cierra.

```
public class holamundo {
......
}
```

- La función o método *main*

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

El código Java en las clases se agrupa en métodos o funciones. Cuando Java va a ejecutar el código de una clase, lo primero que hace es buscar el método *main* de dicha clase para ejecutarlo.

El método **main** tiene las siguientes particularidades:

- Es público (**public**). Esto es así para poder llamarlo desde cualquier lado.
- Es estático (**static**). Al ser *static* se le puede llamar sin tener que instanciar la clase.
- No devuelve ningún valor (modificador **void**).
- Admite una serie de parámetros (**String [ ] args**) que en este ejemplo concreto no son utilizados.

Como puede verse en el ejemplo, el método *main* abarca todo el código contenido entre las llaves.

Mostrar texto por pantalla.

Parece intuitivo saber que el texto se mostrará por pantalla ejecutando la siguiente línea:

```
System.out.println ("Hola Mundo");
```

Para sacar información por pantalla en Java se utiliza la clase **System** que puede ser llamada desde cualquier punto de un programa, la cual tiene un atributo **out** que a su vez tiene dos métodos muy utilizados: **print()** y **println()**. La diferencia entre estos dos últimos métodos es que en el segundo se añade un retorno de línea al texto introducido. Como se puede ver la orden termina en ; (todas las ordenes en Java terminan en ; salvo los cierres de llaves a los cuales no hace falta ponérselo pues se sobreentiende que se finaliza la orden).

1.3 ENTORNOS INTEGRADOS DE DESARROLLO

Un IDE o Entorno Integrado de Desarrollo es una herramienta con el cual poder desarrollar y probar proyectos en un lenguaje determinado.

Recuerda
JDK o Java Development Kit es el software necesario para poder desarrollar y ejecutar programas java. También se denomina SDK (*Standard Development Kit*) o incluso J2SE (*Java 2 platform Standard Edition*).

Lo primero que hay que hacer cuando se instala un IDE es configurar como mínimo la ruta del JDK (*Java Development Kit*). Si no se tiene el JDK no se podrá trabajar con Java, luego habrá que instalarlo primero. En Ubuntu Linux basta con ejecutar desde consola el siguiente comando:

```
$ sudo apt-get install sun-java6-jdk
```


Importante
Cuando se instale un entorno integrado de desarrollo hay que asegurarse que las opciones que indican las rutas de las bibliotecas, el JDK y demás recursos son correctas. Si no se hace esto el programa nunca podrá ejecutar ni compilar programas.

Una buena opción para empezar a programar en Java es instalar Geany. Geany es un IDE muy liviano y muy intuitivo y su instalación es sumamente sencilla. En Ubuntu Linux se instala ejecutando desde consola el siguiente comando:

```
$ sudo apt-get install geany
```

Una vez instalado el programa, hay que configurar la variable PATH en Windows® (Panel de control -> sistema -> Opciones Avanzadas -> variables de entorno) o las variables JAVA_HOME y JAVA en Linux.

¿Necesitas ayuda para instalar Geany en tu equipo?
En el material adicional del libro tienes un manual paso a paso para instalar Geany y el JDK en Windows® y en Linux.

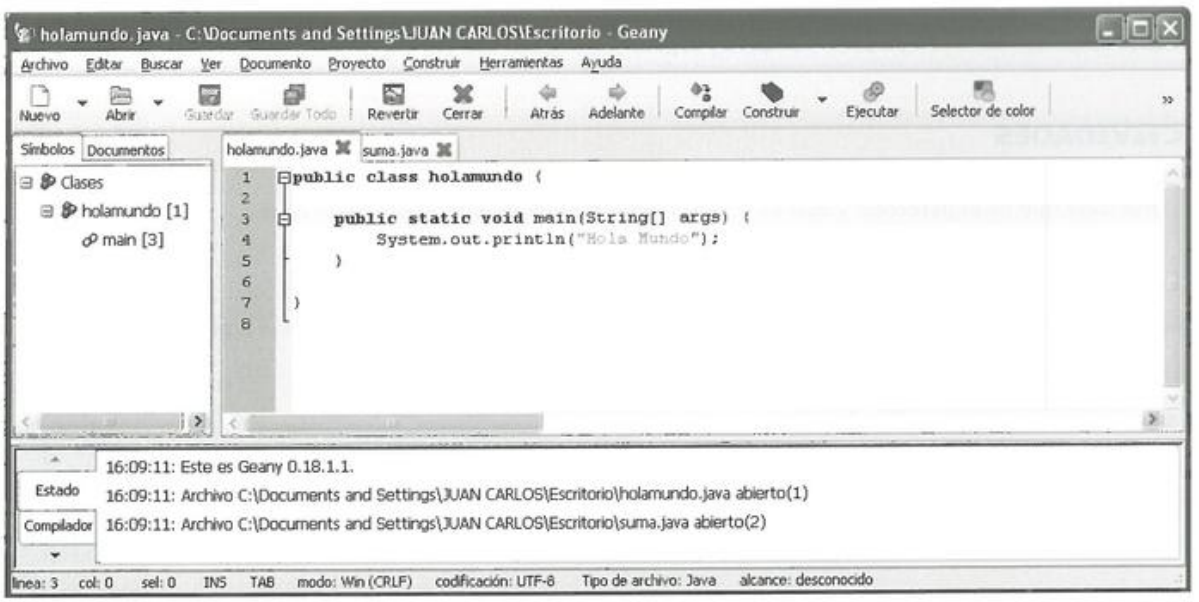


Figura 1.5. Geany. Un entorno de desarrollo ligero y versátil

La secuencia de creación y ejecución de un programa en Java es un proceso que sigue los siguientes pasos: EDITAR →GUARDAR →COMPILAR →EJECUTAR.

Existen muchos IDE para trabajar con Java. En todos los ejemplos se ha utilizado Geany pero si se quiere algo mas potente, una buena opción es Eclipse. Eclipse fue desarrollado primeramente por IBM, aunque actualmente es un IDE de código abierto desarrollado y mantenido por la Fundación Eclipse (http://www.eclipse.org/). Eclipse puede utilizarse para Java y añadiendo plugins pueden utilizarse otros lenguajes de programación. Eclipse ha desarrollado numerosas versiones, todas con nombres estelares (Callisto, Europa, Ganymede, Galileo, Helios...). Otra opción no

menos interesante; es NetBeans de la extinta SUN® ahora Oracle®. NetBeans es una aplicación de código abierto y muchos desarrolladores Java la utilizan. Como consejo, se recomienda Geany para pequeños proyectos y programas como NetBeans o Eclipse para proyectos más serios.

Recuerda
NetBeans y Eclipse son entornos de desarrollo libres.

¿Es necesario un IDE para compilar y ejecutar Java?

La respuesta es No. No es necesario compilar desde un IDE nuestro programa. En principio, si la variable PATH está correctamente configurada bastaría con ejecutar desde línea de comando y desde el mismo directorio donde se encuentra el fichero holamundo.java el siguiente comando:

```
$javac holamundo.java
```

Si el programa está correctamente escrito y el compilador no muestra ninguna salida de error aparecerá un fichero holamundo.class que será el bytecode o código que podrá ser ejecutado en cualquier máquina virtual Java.

ACTIVIDADES

- Investiga qué es el bytecode y qué es y cómo funciona una máquina virtual de Java.

Una vez tenemos el fichero .class hay que ejecutar el programa. Esto se realiza con el siguiente comando:

```
$java holamundo
```

O si se está en un entorno Windows®:

```
C:\> java holamundo
```

Y saldrá en la pantalla la cadena que queremos "Hola Mundo".

ACTIVIDADES

- Instala Geany y el JDK en tu máquina. Una vez instaladas estas dos cosas configura las variables PATH y CLASSPATH en la máquina.
- Prueba a compilar dentro y fuera del IDE el programa holamundo y comprueba que funciona ejecutándolo.

1.4 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:

Tabla 1.1. Tipos de datos simples

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 booleanas es verdadero (true) o falso(false).

ACTIVIDADES

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

1.4.1 ¿CÓMO SE UTILIZAN LOS TIPOS DE DATOS?

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

Tabla 1.2. Utilización de tipos de datos

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 decimal 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; /* 1x106 */ double medio1/2D; /*0.5 la D en este caso indica Double*/

1.5 CONSTANTES Y LITERALES

1.5.1 LAS CONSTANTES

Cuestión de estilo
Las constantes se declaran en mayúscula mientras que las variables se hacen en minúscula (esto se realiza como norma de estilo).

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;
```

Importante
Las constantes se utilizan en datos que nunca varían (IVA, PI, etc.). Utilizando constantes y no variables nos aseguramos 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).

1.5.2 LOS LITERALES

Un literal puede ser una expresión:

- De tipo de dato simple.
- El 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 estructurada".

1.6 VARIABLES

Una variable no es 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 utilizados 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.

Inicialización de variables
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 se inicializan por defecto.

Importante
Una variable local no puede ser declarada como *static*.

1.6.1 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.

Recuerda
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.

1.7 OPERADORES Y EXPRESIONES

1.7.1 OPERADORES ARITMÉTICOS

Los operadores aritméticos son utilizados para realizar operaciones matemáticas.

Tabla 1.3. Operadores aritméticos

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

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
```

1.7.2 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 que son los operandos como se puede ver:

Tabla 1.4. Operadores relacionales

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

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
```

1.7.3 OPERADORES LÓGICOS

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

Tabla 1.5. Operadores lógicos

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.

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
```

1.7.4 OPERADORES UNITARIOS O UNARIOS

Tabla 1.6. Operadores unitarios

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)

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

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

1.7.5 OPERADORES DE BITS

Tabla 1.7. 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.

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
```

1.7.6 OPERADORES DE ASIGNACIÓN

Tabla 1.8. 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.

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
```

1.7.7 PRECEDENCIA DE OPERADORES

Consejo
Utiliza paréntesis y 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:

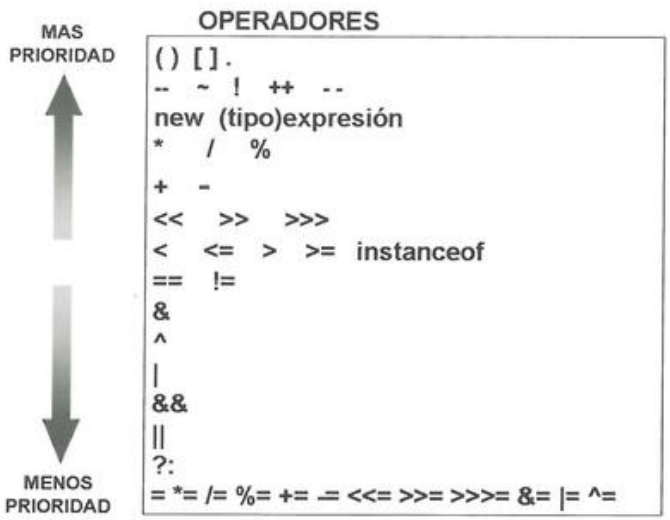


Figura 1.6. Prioridad de los operadores

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.

1.8 CONVERSIONES DE TIPOS (CAST)

Existen dos tipos de conversiones, las conversiones explícitas e implícitas.

- **Conversiones implícitas**. Se realiza de forma automática entre dos tipos de datos diferentes. Requiere que la variable destino (la colocada a la izquierda) tenga más precisión que la variable origen (situada a la derecha).

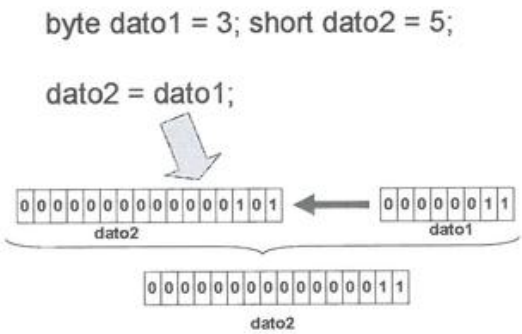


Figura 1.7. Ejemplo de conversión implícita

- **Conversiones explícitas**. En este caso es el programador el que fuerza la conversión mediante una operación llamada *cast* con el formato:

  (tipo) expresión

Recuerda
Como puede ser comprensible no se pueden realizar conversiones entre enteros y *booleanos* o reales y *booleanos*.

Un ejemplo de conversión explícita sería el siguiente:

```
int idato=5;
byte bdato;
bdato = (byte)idato;
System.out.println(bdato); // sacará 5 por pantalla
```


Consejo
Intenta evitar las conversiones de tipos en la medida de lo posible. En algunas conversiones explícitas como ya supondrás pueden perder información en algunos casos.

RESUMEN DEL CAPÍTULO

En este tema se introduce al alumno en los lenguajes de programación, y más concretamente, al lenguaje Java. Este tema es una primera toma de contacto del alumno con la programación y se hace desde una posición global viendo cómo funciona un programa muy sencillo (el famoso Hola Mundo) y desde los aspectos básicos como son las variables, los tipos de datos, los comentarios, los operadores, etc.

Es posible que el alumno en estos primeros temas vea conceptos que luego se estudien con más profundidad en apartados siguientes. El alumno en este tema deberá de entender la estructura, cómo funciona Java y poner énfasis en el estudio de los aspectos básicos del lenguaje, los distintos tipos de datos y en la utilidad y uso de los operadores y expresiones.

Tambien es importante para el capítulo que el alumno sepa instalar y utilizar un IDE. La instalación, compilación y ejecución de programas es una cuestión básica que debe de manejar el alumno para el resto del libro.

EJERCICIOS RESUELTOS

1. Realiza un método para la clase *Test* que genere letras de forma aleatoria. Como ejercicio complementario investiga el funcionamiento y uso de la función Math.random().

Solución:

```
class Test {
    public static char getLetras(){
    return (char)(Math.random()*26 + 'a');
    }
    public static void main(String[] args) {
    System.out.println(getLetras());
    System.out.println(getLetras());
    System.out.println(getLetras());
    System.out.println(getLetras());
    }
}
```

2. El objetivo de este ejercicio es cumplimentar la segunda columna de la siguiente tabla. Como puedes observar ya está cumplimentada pero en los ejercicios propuestos tendrás que comprobar y cumplimentarla tú.

Tabla 1.9. Tabla ejercicio 2

¿Compilará y funcionará el siguiente código?		En caso afirmativo explica que mostrará por pantalla. En caso negativo explica por qué no funciona.
int a = 'a'; System.out.println(a);	☒ Funciona ☐ No funciona	El código introduce 97 en la variable a que es el valor del código ASCII 'a' y lo muestra por pantalla.
int pi = 3.14; System.out.println(pi);	☐ Funciona ☒ No funciona	No funciona. No es posible introducir un número real en una variable de precisión entero
double pi = 3,14; System.out.println(pi);	☐ Funciona ☒ No funciona	Para que funcione basta con cambiar la coma por un punto.
boolean adivina = (1 == 4); System.out.println(adivina);	☒ Funciona ☐ No funciona	Correcto. Muestra false por pantalla porque los dos valores no son iguales.
boolean adivina = (97 == 'a' == 97); System.out.println(adivina);	☐ Funciona ☒ No funciona	No funcionará porque la primera parte de la comparación 97 == 'a' genera un booleano, y al comparar un booleano con un entero (97) el compilador dará un error.
boolean adivina = (97 == 'a' == true); System.out.println(adivina);	☒ Funciona ☐ No funciona	Muestra true por pantalla porque 97 es el código ASCII de 'a' y por lo tanto dará true. Al comparar true con otro valor booleano como true el resultado será true.

3. Averigua si las siguientes afirmaciones son verdaderas o falsas:

- En Java generalmente un programa consta de varias clases las cuales se compilan en un único fichero.
- El método *main* puede ser *static* o no. En caso de no ser *static* puede haber varios en un mismo programa.
- Los métodos y funciones difieren en Java en que en los primeros no devuelven ningún valor.
- Es posible hacer **byte a = 200;**. El único problema es que como una variable byte solamente almacena hasta el valor 127 la variable a valdrá solo 127.

La solución a este ejercicio está al final de los ejercicios propuestos.

4. Realiza un programa en Java que dada dos variables a y b, intercambie los valores de a y b.

Solución:

```
class intercambio {
    public static void main(String[] args) {
        int a= 5, b= 8;
        int tmp;

        tmp=a;
        a=b;
        b=tmp;
        System.out.println("El valor de a ahora es: "+a);
        System.out.println("El valor de b ahora es: "+b);
    }
}
```

5. Dentro de una clase joven tenemos las variables enteras *edad*, *nivel_de_estudios* e *ingresos*.

Necesitamos almacenar en la variable *booleana jasp* el valor:

- Verdadero. Si la edad es menor o igual a 28, el nivel_de_estudios es mayor que tres y los ingresos superan los 28.000 (euros).
- Falso. En caso contrario.

Escribe el código necesario (2 líneas).

Solución:

```
jasp = false;
jasp = ((edad <= 28) && (nivel_de_estudios > 3) && (ingresos > 28000));
```

6. ¿Que mostrará este programa por pantalla?

```
public class Test {
   public static void main(String[] args) {
       int i=0x100;
       i >>>= 1;
       System.out.println(i);
   }
}
```

Solución:

128

EJERCICIOS PROPUESTOS

1. Modifica el siguiente programa para hacer que compile y funcione:

```
class suma
{
    static int n1=50;
    public static void main(String [] args)
    {
    int  n2=30, suma=0, n3;
    suma=n1+n2;
    System.out.println("LA SUMA ES: " + suma);
    suma=suma+n3;
    System.out.println(suma);
    }
}
```

2. ¿Por qué no compila el siguiente programa? Modifícalo para hacer que funcione.

```
class suma
{
    public static void main(String [] args)
    {
    int  n1=50,n2=30,
    boolean suma=0;
    suma=n1+n2;
    System.out.println("LA SUMA ES: " + suma);
    }
}
```

3. El siguiente programa tiene 3 fallos, averigua cuáles son y modifica el programa para que funcione.

```
class cuadrado
{
    public static void main(String [] args)
    {
    int  numero=2,
    cuad=numero * número;
    System.out.println("EL CUADRADO DE "+NUMERO+" ES: " + cuad);
```

```
    }
}
```

- **4.** ¿Qué mostrará el siguiente código por pantalla?

```
int  num=5;
num += num - 1 * 4 + 1;
System.out.println(num);
num=4;
num %= 7 * num % 3 * 7 >> 1;
System.out.println(num);
```

- **5.** Realiza un programa que calcule la longitud de una circunferencia de radio 3 metros.
- **6.** Realiza un programa que calcule el área de una circunferencia de radio 5,2 centímetros.
- **7.** Realiza un programa que muestre en pantalla, respetando los retornos de línea, el siguiente texto:

  Me gusta la programación

  cada día más.

- **8.** (Ejercicio de dificultad alta) Realiza un programa que genere letras aleatoriamente y determine si son vocales o consonantes.
- **9.** Cumplimenta la siguiente tabla:

Tabla 1.10. Tabla ejercicio 9

¿Compilará y funcionará el siguiente código?		En caso afirmativo explica qué mostrará por pantalla. En caso negativo explica por qué no funciona.
boolean adivina = ((97 == 'a') && true); System.out.println(adivina);	☐ Funciona ☐ No funciona	

int a=1;
int b = a>>>2;
System.out.println(b);
	☐ Funciona
	☐ No funciona	

int a = 7 | 4;
System.out.println(a);
int b = 3 | 4;
System.out.println(b);
	☐ Funciona
	☐ No funciona	

int a = 7 & 4;
System.out.println(a);
int b = 3 & 4;
System.out.println(b);
	☐ Funciona
	☐ No funciona	

int a = ~4; System.out.println(a);	☐ Funciona ☐ No funciona	
int a = (~4 * 5)&1; System.out.println(a);	☐ Funciona ☐ No funciona	

10. Dentro de una clase *joven* tenemos las variables enteras *edad*, *nivel_de_estudios* e *ingresos*.

Necesitamos almacenar en la variable *booleana* jasp el valor:

- Verdadero. Si la *edad* es menor o igual a 28 y el *nivel_de_estudios* es mayor que tres, o bien, la *edad* es menor de 30 y los *ingresos* superan los 28.000 (euros).
- Falso. En caso contrario.

Escribe el código necesario (2 líneas).

11. Realiza un programa con una variable entera **t** la cual contiene un tiempo en segundos y queremos conocer este tiempo pero expresado en horas, minutos y segundos.

12. (Ejercicio de dificultad alta) Realiza un programa que dado un importe en euros nos indique el mínimo número de billetes y la cantidad sobrante que se pueden utilizar para obtener dicha cantidad.

Por ejemplo:

232 euros:

1 billete de 200.

1 billete de 20.

1 billete de 10

Sobran 2 euros.

Solución al ejercicio resuelto número 3:

Todas las afirmaciones son falsas.

2 Programación orientada a objetos. Objetos

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las características básicas de la programación orientada a objetos.
- ✓ Valorar las ventajas de la programación orientada a objetos frente a la tradicional.
- ✓ Conocer la estructura básica de las clases.
- ✓ Reconocer las diferencias entre un objeto y una clase. Cómo se inicializan y cómo finalizan.
- ✓ Diseñar clases sencillas.
- ✓ Organizar clases en paquetes y utilizar los mismos.
- ✓ Programar y diseñar métodos que acepten parámetros.

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 programación de última generación permiten la programación orientada a objetos, así como también la programación clásica, con lo cual puede entenderse la POO como una evolución de la programación clásica.

2.1 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 su caso Felipe), un color de plumaje (verde), una edad (en su caso 2 años) y si son domésticos o no (Felipe sí es doméstico pues lo tengo ahora en mi hombro).

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, 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 ha, 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 de cero a muchos atributos) y los métodos (una clase puede tener de cero a muchos métodos). Los métodos para la gente que haya programado en cualquier lenguaje de programación son los llamados procedimientos o funciones.

Recuerda
El nombre de la clase y del fichero que la contiene deberá de ser el mismo.

Objetos

Un objeto tiene una serie de características como son las siguientes:

- **Identidad**. Cada objeto es único y diferente de otro objeto. Mi loro Felipe es diferente a otros loros aunque estos sean verdes, tengan 2 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.).

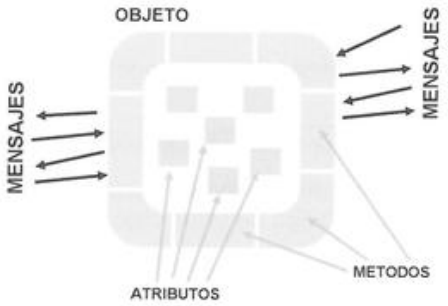


Figura 2.1. Estructura de un objeto

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 procesándose la información para luego destruirse y liberar la memoria que estaban ocupando.

A FONDO

LOS OBJETOS

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 de 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, sino que tenemos que reemplazar el objeto por otro similar, o bien reprogramarlo.
- **Ocultación de información**. En la POO se ocultan los detalles de implementación y lo que priva es la interfaz.

2.2 CARACTERÍSTICAS DE LA PROGRAMACIÓN ORIENTADA 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 tomar 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 ejecuta.

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 y el interior de los métodos que permiten manipularlos, solamente tienen que conocer su interfaz.

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

Figura 2.2. 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.

Importante
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.

2.3 PROPIEDADES Y MÉTODOS DE LOS OBJETOS

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 de conocer la clase *test* son los métodos públicos para utilizar la clase.

2.4 PROGRAMACIÓN DE LA CONSOLA: ENTRADA Y SALIDA DE INFORMACIÓN

Java permite la entrada por teclado y la salida de información a través de la pantalla mediante la clase *System* del paquete java.lang. La clase *System* contiene, entre otros tres objetos los cuales están asociados a tres flujos estándar que se abren cuando se ejecuta el programa y se cierran cuando éste finaliza. Estos objetos *static* son los siguientes:

- **System.out**. Referencia a la salida estándar (pantalla).
- **System.in**. Referencia a la entrada estándar (teclado). El objeto *System.in* pertenece a la clase *InputStream*. Un ejemplo de utilización es el siguiente:

```
char c;
try{
c = (char) System.in.read();
}catch(Exception e){
 e.printStackTrace();
}
```

Recuerda
El método **void printStackTrace()** indica el método donde se lanzó la excepción.

En el ejemplo anterior se llama al método *read()* del objeto *System.in* que es el método básico de lectura de un carácter por teclado. Generalmente, lo que se hace es leer una línea completa desde teclado, no un carácter nada más. Por lo tanto hay que utilizar el siguiente código:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader buff = new BufferedReader(isr);
String ln = buff.readLine();
```

En este código se crea un objeto del tipo InputStreamReader al cual se le pasa como parámetro en el constructor el objeto System.in (que es un *InputStream*). Es una clase derivada de *Reader* y acepta en el constructor como parámetro un InputStream. En la siguiente línea, el objeto *buff* va a pedir un *reader* al crearse y aprovechamos para pasarle como parámetro el *InputStreamReader* (isr) que hemos creado anteriormente.

Truco
Otra forma de leer datos desde teclado es el siguiente:

```
String sdato = System.console().readLine(); //lectura de un string.
dato = Integer.parseInt(sdato); //transformación del string en entero.
```

- **System.err**. Referencia a la salida de error estándar que, al igual que la salida estándar, es la pantalla. Utilizado para mostrar mensajes de error. Al igual que System.out tiene los métodos *print()* y *println()*. Un ejemplo de utilización de este objeto es el siguiente:

```
int a=10, b=0, c;

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

2.5 PARÁMETROS Y VALORES DEVUELTOS

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 2.3. Parámetros y valores devueltos por un método

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*.

2.6 CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

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++ 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.

Importante: Destructores en Java
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. Es posible crear objetos de la clase *pájaro* de distintas formas.

2.7 USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

En este apartado se va a ver cuándo un método o atributo debe de 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 como 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, el método *nuevopajaro()*, *muestrapajaros()* o el método *main()* tienen 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();
        }
    }
```

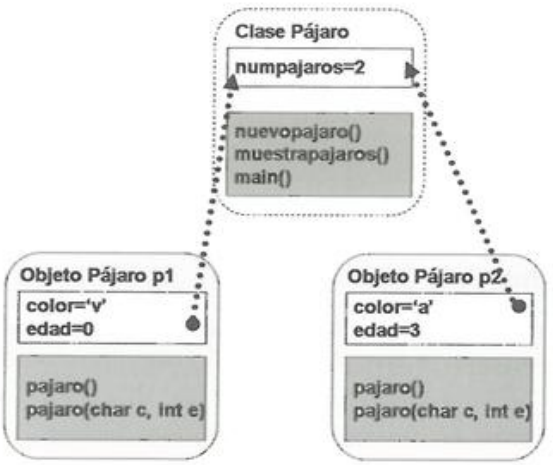


Figura 2.4. Clase pájaro y objetos de dicha clase

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*.

2.8 LIBRERÍAS DE OBJETOS (PAQUETES)

Recuerda
Librería o paquete es lo mismo.

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, estarán 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.

A FONDO

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 bién podemos importar todas las clases de un paquete:

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

En este caso podremos utilizar todas las clases del paquete. 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:

```
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 no se utiliza esta última opción porque hay que escribir mucho código.

Un applet
Un *applet* es una aplicación Java que se ejecuta en la ventana de un navegador. Los *applets* se ejecutan en la máquina del cliente y nunca en el servidor.

Tabla 2.1. Librerías Java

Paquete o librería	Descripción
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.util	Librería con clases de utilidad general para el programador.
java.applet	Librería para desarrollar *applets*.
java.awt	Librerías con componentes para el desarrollo de interfaces de usuario.
java.swing	Librerías con componentes para el desarrollo de interfaces de usuario. Similar al paquete awt.
java.net	En combinación con la librería java.io, va a permitir crear aplicaciones que realicen comunicaciones con la red local e Internet.
java.math	Librería con todo tipo de utilidades matemáticas.
java.sql	Librería especializada en el manejo y comunicación con bases de datos.
java.security	Librería que implementa mecanismos de seguridad.
java.rmi	Paquete que permite el acceso a objetos situados en otros equipos (objetos remotos).
java.beans	Librería que permite la creación y manejo de componentes *javabeans*.

Un javabean
Un *javabean* es un componente reutilizable que encapsula varios objetos en uno solo. *Bean* es una vaina en inglés y, como su nombre indica, permite tener un único objeto en vez de varios más simples.

2.8.1 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 de 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 ya 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®.

RESUMEN DEL CAPÍTULO

En este tema se va a abordar las características básicas de la programación orientada a objetos. El objetivo del mismo es comenzar a trabajar con la orientación a objetos mediante las características más básicas de la misma. El alumno deberá de poner interés en el aprendizaje de estos conceptos que serán desarrollados más en profundidad en los capítulos 4 y 5 de este libro. Una vez trabajado el tema, el alumno será capaz de resolver problemas básicos utilizando clases con métodos sencillos.

EJERCICIOS RESUELTOS

1. Realiza una clase *Temperatura*, la cual convierta grados Celsius a Farenheit y viceversa. Para ello crea dos métodos *double celsiusToFarenheit(double)* y *double farenheitToCelsius(double)*.

En la construcción ten en cuenta las siguientes fórmulas:

- Fahrenheit a Celsius $C = (F - 32)/1,8$
- Celsius a Fahrenheit $F = (1,8)C + 32$

Solución:

```
class Temperatura {
    public static double celsiusToFarenheit(double temp)
    {
          return (1.8)*temp + 32;
    }
    public static double farenheitToCelsius(double temp)
    {
          return (temp - 32)/1.8;
    }
    public static void main(String[] args) {
    System.out.println("0 grados Celsius son "+celsiusToFarenheit(0)+" Grados
Farenheit");
    System.out.println("15 grados Celsius son "+celsiusToFarenheit(15)+" Grados
Farenheit");
    System.out.println("20 grados Celsius son "+celsiusToFarenheit(20)+" Grados
Farenheit");
    System.out.println("0 grados Farenheit son "+farenheitToCelsius(0)+" Grados
Celsius");
    System.out.println("40 grados Farenheit son "+farenheitToCelsius(45)+" Grados
Celsius");
    System.out.println("70 grados Farenheit son "+farenheitToCelsius(70)+" Grados
Celsius");
   }
}
```

El programa anterior dará la siguiente salida:

0 grados Celsius son 32.0 grados Farenheit.

15 grados Celsius son 59.0 grados Farenheit.

20 grados Celsius son 68.0 grados Farenheit.

0 grados Farenheit son -17.77777777777778 grados Celsius.

40 grados Farenheit son 7.222222222222222 grados Celsius.

70 grados Farenheit son 21.11111111111111 grados Celsius.

2. Tenemos la siguiente clase *coche*:

```
class coche {
   private int velocidad;
   coche (){velocidad=0;}
}
```

Añade a la clase *coche* los siguientes métodos:

int getVelocidad(). Este método devuelve la velocidad actual.

void acelera(int mas). Este método actualiza la velocidad a mas kilómetros más.

void frena(int menos). Este método actualiza la velocidad a menos kilómetros menos.

Solución:

```
class coche {
   private int velocidad;
   coche (){velocidad=0;}
   public int getVelocidad(){return velocidad;}
   public void acelera(int mas){velocidad+=mas;}
   public void frena(int menos){velocidad-=menos;}
}
```

3. Averigua si son verdaderas o falsas las siguientes afirmaciones:

1. Las clases que forman parte de un paquete deben derivar todas ellas de una misma superclase.
2. En una clase *Java* los atributos pueden ser tipos primitivos o bien pueden ser objetos de otra clase.
3. Cuando se escribe un programa o aplicación orientada a objetos, 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.
4. La ocultación de información significa que al proporcionar los ficheros .class, el programador no tiene por qué proporcionar los ficheros .java donde reside todo el código.
5. Un paquete o *package* es un conjunto de clases relacionadas entre sí, las cuales están ordenadas de forma arbitraria.
6. Un *javabean* es un componente reutilizable que encapsula varios objetos en uno solo.
7. El método *void printStackTrace()* indica el método donde se lanzó la excepción.
8. La abstracción en POO es una de las propiedades fundamentales de la misma mediante la cual los objetos se ven según su comportamiento externo.

9. El polimorfismo permite crear varias formas del mismo método, de tal manera que un mismo método ofrezca comportamientos idénticos pero con distinta forma.
10. Las clases se escriben en ficheros ASCII. El nombre del fichero puede ser cualquiera, pero lo importante es que la extensión sea .java.
11. En un programa orientado a objetos primeramente se crean los objetos y entre ellos se envían mensajes procesándose la información, para luego destruirse y liberar la memoria que estaban ocupando.

4. ¿Está correctamente definida la siguiente clase? ¿Compilará o habrá que modificarla para poder generar el fichero .class?

```
class pajaro {
    public void setEdad(int e){edad = e;}
    public void printEdad(){System.out.println(edad);}
    public void setcolor(char c){color=c;}
    private char color;
    private int edad;
}
```

Solución:

Realmente la clase compila y funcionará sin problemas. No obstante, los atributos se suelen colocar por convenio en la parte superior del cuerpo de la clase y los métodos en la parte inferior. No es común encontrarse los atributos al final.

5. La siguiente clase tiene problemas de compilación:

```
public class satelite {
    private double meridiano;
    private double paralelo;
    private double distancia_tierra;
    satelite (double m,double p,double d){
          meridiano=m;
          paralelo=p;
          distancia_tierra=d;
    }
    satelite (){
          meridiano=paralelo=distancia_tierra=0;
    }
    public void setPosicion(double m,double p,double d){
          meridiano=m;
          paralelo=p;
          distancia_tierra=d;
    }
    public void printPosicion(){
```

```
            System.out.println(«El satélite se encuentra en el paralelo "+ paralelo+"
Meridiano "+meridiano+" a una distancia de la tierra de "+distancia_tierra+
"Kilómetros");
    }
}
```

Averigua los problemas y corrígelos.

Solución:

La solución se encuentra al final de los ejercicios propuestos.

6. Crea una clase *rebajas* con un método *descubrePorcentaje()* que descubra el descuento aplicado en un producto. El método recibe el precio original del producto y el rebajado y haya el porcentaje.

Solución:

```
public class rebajas {
    public static double descubrePorcentaje(double original, double actual){
            return(original-actual)*100/original;
    }
    public static void main(String[] args) {
            System.out.println(descubrePorcentaje(100,79));
            System.out.println(descubrePorcentaje(100,50));
    }
}
```


EJERCICIOS PROPUESTOS

1. Realiza una clase *finanzas* que convierta dólares a euros y viceversa. Codifica los métodos *dolaresToEuros* y *eurosToDolares*. Prueba que dicha clase funciona correctamente haciendo conversiones entre euros y dólares. La clase tiene que tener:
   - Un constructor *finanzas()* por defecto el cual establecerá el cambio Dólar-Euro en 1.36.
   - Un constructor *finanzas(double)*, el cual permitirá configurar el cambio dólar-euro.

2. Realiza una clase *minumero* que proporcione el doble, triple y cuádruple de un número proporcionado en su constructor (realiza un método para doble, otro para triple y otro para cuádruple). Haz que la clase tenga un método *main* y comprueba los distintos métodos.

- 3. Realiza una clase *número* que almacene un número entero y tenga las siguientes características:
  - Constructor por defecto que inicializa a 0 el número interno.
  - Constructor que inicializa el número interno.
  - Método aniade que permite sumarle un número al valor interno.
  - Método resta que resta un número al valor interno.
  - Método getValor. Devuelve el valor interno.
  - Método getDoble. Devuelve el doble del valor interno.
  - Método getTriple. Devuelve el triple del valor interno.
  - Método setNumero. Inicializa de nuevo el valor interno.

- 4. Empareja cada paquete con su correspondiente descripción:

Tabla 2.2. Tabla ejercicio 4

Paquete o librería	Descripción
java.security	Paquete con clases esenciales de Java. No hace falta ejecutar la sentencia import para utilizar sus clases. Librería por defecto.
java.rmi	Librería especializada en el manejo y comunicación con bases de datos.
java.beans	Librería con clases de utilidad general para el programador.
java.applet	Librería para desarrollar *applets*.
java.math	Librería que implementa mecanismos de seguridad.
java.sql	Paquete que permite el acceso a objetos situados en otros equipos (objetos remotos).
java.net	Librería de Entrada/Salida. Permite la comunicación del programa con ficheros y periféricos.
java.awt	Librerías con componentes para el desarrollo de interfaces de usuario.
java.swing	Librería que permite la creación y manejo de componentes *javabeans*.
java.io	Librerías con componentes para el desarrollo de interfaces de usuario. Similar al paquete awt.
java.lang	En combinación con la librería java.io, va a permitir crear aplicaciones que realicen comunicaciones con la red local e Internet.
java.util	Librería con todo tipo de utilidades matemáticas.

- **5.** Modificación del ejercicio resuelto número 5.
  - Modifica la clase *satélite* y añádele los siguientes métodos:
    - Método *void variaAltura(double desplazamiento)*. Este método acepta un parámetro que será positivo o negativo dependiendo de si el satélite tiene que alejarse o acercarse a La Tierra.
    - Método *boolean enOrbita()*. Este método devolverá *false* si el satélite está en tierra y *true* en caso contrario.
    - Método *void variaPosicion(double variap, double variam)*. Este método permite modificar los atributos de posición (meridiano y paralelo) mediante los parámetros variap y variam. Estos parámetros serán valores positivos o negativos relativos que harán al satélite modificar su posición.

- **6.** (Ejercicio de dificultad alta) Crea la clase *peso*, la cual tendrá las siguientes características:
  - Deberá tener un atributo donde se almacene el peso de un objeto en kilogramos.
  - En el constructor se le pasará el peso y la medida en la que se ha tomado ('Lb' para libras, 'Li' para lingotes, 'Oz' para onzas, 'P' para peniques, 'K' para kilos, 'G' para gramos y 'Q' para quintales).
  - Deberá de tener los siguientes métodos:
    - getLibras. Devuelve el peso en libras.
    - getLingotes. Devuelve el peso en lingotes.
    - getPeso. Devuelve el peso en la medida que se pase como parámetro ('Lb' para libras, 'Li' para lingotes, 'Oz' para onzas, 'P' para peniques, 'K' para kilos, 'G' para gramos y 'Q' para quintales).
  - Para la realización del ejercicio toma como referencia los siguientes datos:
    - 1 Libra = 16 onzas = 453 gramos.
    - 1 Lingote = 32,17 libras = 14,59 kg.
    - 1 Onza = 0,0625 libras = 28,35 gramos.
    - 1 Penique = 0,05 onzas = 1,55 gramos.
    - 1 Quintal = 100 libras = 43,3 kg.
  - Crea además un método *main* para testear y verificar los métodos de esta clase.

- **7.** Crea una clase con un método *millasAMetros()* que toma como parámetro de entrada un valor en millas marinas y las convierte a metros.
  - Una vez tengas este método escribe otro *millasAKilometros()* que realice la misma conversión, pero esta vez exprese el resultado en kilómetros.

  Nota: 1 milla marina equivale a 1852 metros.

- **8.** Crea la clase *coche* con dos constructores. Uno no toma parámetros y el otro sí. Los dos constructores inicializarán los atributos marca y modelo de la clase. Crea dos objetos (cada objeto llama a un constructor distinto) y verifica que todo funciona correctamente.

- **9.** Implementa una clase *consumo*, la cual forma parte de la centralita electrónica de un coche y tiene las siguientes características:
  - Atributos:

- kms. Kilómetros recorridos por el coche.
- litros. Litros de combustible consumido.
- vmed. Velocidad media.
- pgas. Precio de la gasolina.

- Métodos:
  - getTiempo. Indicará el tiempo empleado en realizar el viaje.
  - consumoMedio. Consumo medio del vehiculo (en litros cada 100 kilómetros).
  - consumoEuros. Consumo medio del vehiculo (en euros cada 100 kilómetros).

No olvides crear un constructor para la clase que establezca el valor de los atributos. Elige el tipo de datos más apropiado para cada atributo.

10. Para la clase anterior implementa los siguientes métodos, los cuales podrán modificar los valores de los atributos de la clase:

- setKms
- setLitros
- setVmed
- setPgas

11. (Ejercicio de dificultad alta) El restaurante mejicano de Israel cuya especialidad son las papas con chocos nos pide diseñar un método con el que se pueda saber cuántos clientes pueden atender con la materia prima que tienen en el almacén. El método recibe la cantidad de papas y chocos en kilos y devuelve el número de clientes que puede atender el restaurante teniendo en cuenta que por cada tres personas, Israel utiliza un kilo de papas y medio de chocos.

12. Modifica el programa anterior creando una clase que permita almacenar los kilos de papas y chocos del restaurante. Implementa los siguientes métodos:

- *public void addChocos(int x).* Añade x kilos de chocos a los ya existentes.
- *public void addPapas(int x).* Añade x kilos de papas a los ya existentes.
- *public int getComensales().* Devuelve el número de clientes que puede atender el restaurante (este es el método anterior).
- *public void showChocos().* Muestra por pantalla los kilos de chocos que hay en el almacén.
- *public void showPapas().* Añade Muestra por pantalla los kilos de papas que hay en el almacén.

Solución al ejercicio resuelto número 3:

1. Falsa
2. Verdadera
3. Verdadera
4. Falsa
5. Verdadera
6. Verdadera
7. Verdadera
8. Falsa
9. Falsa
10. Falsa
11. Verdadera

Solución al ejercicio resuelto número 5:

```
public class satelite { //class se escribe con doble s
    private double meridiano;
    private double paralelo; //falta el ; del final
    private double distancia_tierra;
    satelite (double m,double p,double d){
          meridiano=m;
          paralelo=p;
          distancia_tierra=d;
    }
    satelite (){ //cuidado con los acentos
          meridiano=paralelo=distancia_tierra=0;
    }
    public void setPosicion(double m,double p,double d){
//los parámetros siempre separados por comas
          meridiano=m;
          paralelo=p;
          distancia_tierra=d; //variable mal escrita
    }
    public void printPosicion(){
          System.out.println(«El satélite se encuentra en el paralelo «+ paralelo+
« Meridiano «+meridiano+» a una distancia de la tierra de «+distancia_tierra+»
Kilómetros»);
// forma correcta de mostrar el estado del satélite
    }
}
```

3 Estructuras básicas de control

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer y saber aplicar las estructuras básicas del lenguaje de programación Java.
- ✓ Controlar las excepciones básicas que pueda generar un programa.
- ✓ Reconocer la importancia de la documentación en el desarrollo de aplicaciones.
- ✓ Conocer el proceso de prueba y depuración de programas.

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;).

3.1 ESTRUCTURAS DE SELECCIÓN

3.1.1 ESTRUCTURAS IF

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

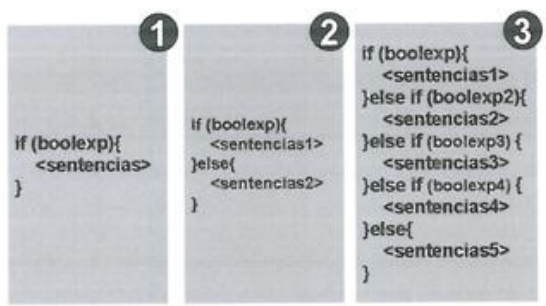


Figura 3.1. Estructura IF

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 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:

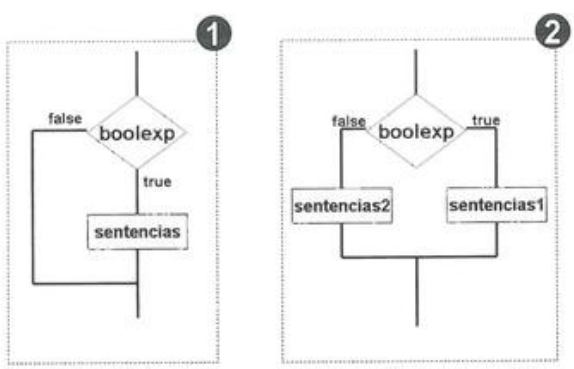


Figura 3.2. Flujo de ejecución en estructuras IF

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");
}
```

3.1.2 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 3.3. Estructura 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;
}
```

3.2 ESTRUCTURAS DE REPETICIÓN

Las estructuras de repetición 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 *booleanas* 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.

3.2.1 BUCLE 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:

```
while (boolexp) {
        sentencias;
}
```

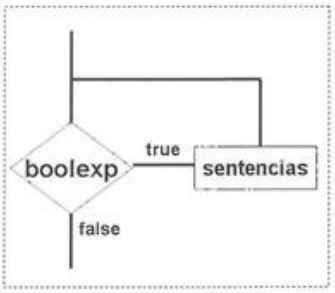


Figura 3.4. Estructura del bucle WHILE

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.

3.2.2 BUCLE DO WHILE

El formato de estructura del bucle **do while** es el siguiente:

```
do{
        sentencias;
} while (expresión);
```

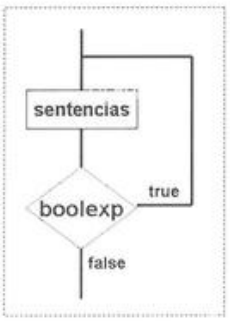


Figura 3.5. Estructura del bucle DO-WHILE

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**.

3.2.3 BUCLE 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;
};
```

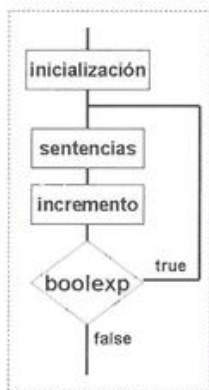


Figura 3.6. Estructura del bucle FOR

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.

3.3 ESTRUCTURAS DE SALTO

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

3.3.1 SENTENCIAS BREAK Y CONTINUE

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.

La sentencia **continue**, por el contrario, solo se utiliza en las estructuras de repetición (bucles) y lo que hace es terminar la iteración i y continúa por la iteración i+1.

Un ejemplo de utilización de **continue** sería el siguiente:

```
int i=0;
while(i<10){ //este programa muestra los números del 1 al 10
       i++;      // sin mostrar el 5
       if (i==5){continue;}
       System.out.println(i);
}
```

3.3.2 SENTENCIAS BREAK Y CONTINUE CON ETIQUETAS

Las sentencias **break** y **continue** con etiquetas mantienen el mismo funcionamiento salvo que en esta ocasión el programador puede controlar qué bucle es el que se deja de ejecutar o en qué bucle continua.

Un ejemplo de **break** con etiquetas es el siguiente:

```
int i=0;
bucleext :
while(i<100){
    i++;
    for (int j=0;j<i;j++){
          System.out.print("*");
          if (i==5){break bucleext;}
    }
    System.out.println("");
}
```

En este código se puede apreciar que solamente se ejecutarán cuatro iteraciones, en la quinta se deja de ejecutar el bucle **while** principal.

Consejo
Ejecuta y estudia detenidamente el siguiente código. Modifícalo para crear la pirámide con más o menos líneas.

En el siguiente ejemplo se combinan las sentencias **break** y **continue** con etiquetas:

```
int i=0;
bucleext :
while(i<20){
      i++;
      for (int k=1;k<(20-i);k+=2){
            if (i%2 == 0){continue bucleext;}
            System.out.print("_");
      }
      for (int j=0;j<i;j++){
            System.out.print("*");
      }
      System.out.println("");
      if (i==19){break bucleext;}
}
```

3.3.3 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 que está ejecutando y permite devolver un valor. Por ejemplo:

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

3.4 CONTROL DE EXCEPCIONES

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

La estructura del control de excepciones tiene el siguiente formato:

Figura 3.7. Control de excepciones, estructura TRY-CATCH-FINALLY

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 éste se ejecutará siempre.

Se va a ver un ejemplo 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 provocando 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 controla la ejecución del programa y éste finalizará de una forma controlada.

3.5 PRUEBA Y DEPURACIÓN DE APLICACIONES

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 permiten **verificar** y **validar** la calidad de un producto software identificando errores de diseño e implementación (que el programa no hace exactamente lo que se pedía o lo hace pero de forma incorrecta).

Se integran dentro de las diferentes fases del ciclo del software y es habitual que dicho proceso de pruebas se inicie desde el mismo momento en que empieza el desarrollo y continúe hasta que finalice el mismo.

Dicha fase ha sido a menudo descuidada y, en ocasiones, casi sacrificada ante las presiones sobre plazos o costes de los proyectos, lo que llevaba a una carencia en la planificación de la misma y una mala documentación desarrollada. Lo ideal es definir un **Plan de Prueba** con una perfecta planificación de tal proceso.

¿Sabías que?
Es habitual que los errores se originen con mayor frecuencia en las primeras fases del desarrollo. Más del 80% de los errores cometidos provienen de las primeras fases del ciclo de vida (análisis de requisitos y diseño funcional y técnico). Además, el coste de corregir dichos errores crece exponencialmente según avanza el proyecto.

Este tipo de pruebas se encargan de ejecutar el software que se está desarrollando o ya está desarrollado bajo condiciones controladas, y 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 pretenden 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).

3.5.1 FALLOS DEL SOFTWARE

Existen un gran número de razones por las que se producen esos errores de programación o *bugs* o esos defectos de forma, todos partiendo de la base de lo complejo que resulta el desarrollo del software. Destacaríamos:

- Poca o falta de comunicación entre diferentes individuos que intervienen en el proceso de desarrollo (cliente, analistas, diseñadores, programadores, etc.).
- Complejidad del software, con poca reutilización de código y que requiere a personas muy expertas.
- Errores de programación. Los programadores son uno de los principales factores. La excesiva confianza, el ego del programador, en ocasiones que lleva a afirmaciones del tipo "no hay problema, es muy fácil" o "puedo terminarlo en pocas horas" en lugar de "es muy complejo, habrá que estudiarlo, puede que cometa errores" o "no sé realmente lo que tardaré".
- Cambios continuos durante el desarrollo del software en cuanto a requerimientos del mismo que conllevan a constantes rediseños y replanificaciones.
- Presiones de tiempos. Conllevan a omitir ciertas fases de pruebas y control.
- Pobre documentación del código. Dificulta la modificación del código el que la documentación sea escasa o de mala calidad.

El **tester** o persona que realiza las pruebas, es habitualmente un profesional de altos conocimientos en lenguajes de programación y métodos, técnicas y herramientas especializadas de pruebas. Se encarga de someter el software a una serie de acciones (diferentes cargas de datos, acciones de entrada, condiciones del sistema, etc.), de forma que éste responde con su comportamiento como reacción. Debe tener una actitud de probar para romper, la habilidad de conseguir el punto de vista del cliente y un buen análisis de detalle para encontrar errores que no se ven a simple vista.

¿Sabías que?
Existe un famoso enunciado cabecera de todo el que lleva a cabo pruebas de software: El *testing* puede probar la presencia de errores pero no la ausencia de ellos (Edsger Dijkstra).

Nunca se debe testear el software en un **entorno de explotación,** sino que deberá probarse en un entorno de pruebas separado físicamente del de producción. Para crear un **entorno de pruebas** en una máquina independiente de la máquina de producción es necesario crear las mismas condiciones que en la máquina de producción por lo que existen herramientas vendidas por los mismos fabricantes de hardware para tal fin.

La mayoría de las grandes organizaciones asumen la responsabilidad del control de calidad y prueba de software de forma que en la producción se suelen incluir equipos dedicados a tal fin. Es habitual que el proceso de pruebas de software sea realizado por un grupo independiente de probadores o *testers* diferente al que participó en su desarrollo.

3.5.2 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 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 chequeos, 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 que la verificación esté completa.

Existen innumerables tipos de pruebas entre las que podemos nombrar: Pruebas de caja negra o caja blanca, unidad de testeo o prueba, integración incremental, pruebas de integración, 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.

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** o **master** se le 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.

¿Sabías que?
Quizás uno de los fallos más históricos causados por un bug en sistemas de computación fue el que se produjo en enero del 2.000 al sufrir las consecuencias del llamado efecto Y2K (Y2K bug).

3.6 DOCUMENTACIÓN DE PROGRAMAS

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.

Toda aplicación se puede contemplar desde dos aspectos: **su descripción física** o **técnica** (cómo es físicamente, analizando los componentes que lo constituyen (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 de cubrir como mínimo ambas facetas, la faceta **técnica** (información para los informáticos) y la **funcional** (información para todos, especialmente para los usuarios).

Hace mucho tiempo empecé a trabajar desarrollando software en grandes proyectos y comencé a ver cómo se aplicaba todo lo visto en la universidad en lo que respecta a la documentación. Yo no pensaba que el proceso de documentación fuese tan importante. 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 una serie de etapas como son las siguientes:

- **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.

- **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.
- **Diseño**. Esta fase consiste en determinar los requisitos generales de la arquitectura de la aplicación y dar una definición precisa de cada subconjunto de la aplicación.
- **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.
- **Pruebas**. En esta fase se realizarán pruebas para garantizar que la aplicación se programó de acuerdo con las especificaciones originales y los distintos programas de los que consta la aplicación están perfectamente integrados y preparados para la explotación.
- **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.
- **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.

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 de 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 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, decir que en cualquier aplicación, **como mínimo**, se deberán de 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 de 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, 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.

RESUMEN DEL CAPÍTULO

En este capítulo se trabaja con las estructuras básicas de control del lenguaje de programación Java (estructuras de selección, repetición, salto, control de excepciones,...) que son la base de todo lenguaje de programación. El alumno, una vez trabajado este tema, podrá abordar y resolver una gran cantidad de problemas. Es importante que el alumno comprenda y valore la importancia de la documentación, prueba y depuración de programas puesto que son pilares básicos de la ingeniería del software y su aplicación es de obligado cumplimiento en proyectos de envergadura.

EJERCICIOS RESUELTOS

1. Transforma el siguiente bucle for en un bucle while:

```
for (i=5; i<15; i++){
System.out.println(i);
}
```

Solución:

```
int i=5;
while (i<15){
        System.out.println(i);
        i++;
}
```

2. Realiza un programa utilizando bucles que muestre la siguiente figura por pantalla:

```
  *
 ***
*****
```

Solución:

```
int i,j,k;
for (i=1;i<=3;i++){ //bucle que cuenta hasta 10
      for (k=1;k<=3-i;k++){
      System.out.print(" ");
      }
      for (j=1;j<=2*(i-1)+1;j++){
      System.out.print("*");
      }
      System.out.println("");
}
```

3. Realiza un programa que muestre por pantalla los 5 primeros números pares.

Solución:

```
int numero = 2;
for (numero=0;numero<=10;numero+=2){
      System.out.println(numero);
}
```

4. Realiza un método que, dado un número de tres cifras, averigüe si es un número Armstrong. Un número es Armstrong cuando la suma de cada uno de los números que lo componen elevado al número de dígitos de dicho número da como resultado el propio número. Como esta definición es algo compleja, con la siguiente imagen se verá más claro qué es un número Armstrong:

Figura 3.8. Comprobación de un número Armstrong

Solución:

```
class armstrong {
    public static int potencia(int base, int exponente){
          int res=base;
          for (int i=0;i<exponente-1;i++){
                res = res * base;
          }
          return res;
    }
```

```
    public static int armstrong(int numero){
            int cifra1 = numero/100;
            int cifra2 = (numero - 100*cifra1)/10;
            int cifra3 = numero - 100*cifra1 - 10*cifra2;
            int dat = potencia(cifra1,3)+potencia(cifra2,3)+potencia(cifra3,3);
            if (dat == numero) return 1;
            return 0;
    }
    public static void main(String[] args) {
            if (armstrong(371)==1) {
                    System.out.println("El número 371 es un número Armstrong");
            }else{
                    System.out.println("El número 371 No es un número Armstrong");
            }
            if (armstrong(423)==1) {
                    System.out.println("El número 423 es un número Armstrong");
            }else{
                    System.out.println("El número 423 No es un número Armstrong");
            }
    }
}
```

- **5. Realiza una clase *letras* que almacene una letra y la convierta a mayúsculas. La clase tendrá los siguientes métodos:**
  - ***getLetra()* el cual devolverá la letra.**
  - ***printLetra()* el cual muestra la letra por pantalla.**

Solución:

```
public class letras {
    private char letra;
    letras(char l){
            letra = l;
            if (l >= 'a'){
                    letra -= 'a';
                    letra += 'A';
            }
    }
    public char getLetra(){return letra;}
    public void printLetra(){
            System.out.println(letra);
    }
}
```

Como ejercicio complementario, modifica el programa anterior para que solamente acepte letras válidas (a-z A-Z). En caso contrario, la clase imprimirá "letra no valida" cuando se invoque el método *printLetra()*.

6. Escribe un programa que cuente en pantalla de 200 a 300.

Solución:

```
public class cuenta {
   public static void main(String[] args) {
          for(int i = 200; i < 301; i++){
                System.out.println(i);
          }
   }
}
```

7. Escribe un programa que cuente por pantalla del 1 al 10 en inglés. Utiliza una estructura switch que incluya la cláusula default.

Solución:

```
public class cuentaingles {
   public static void main(String[] args) {
    for(int i = 1; i < 11; i++) {
          switch(i) {
                case 1: System.out.println("one"); break;
                case 2: System.out.println("two"); break;
                case 3: System.out.println("three"); break;
                case 4: System.out.println("four"); break;
                case 5: System.out.println("five"); break;
                case 6: System.out.println("six"); break;
                case 7: System.out.println("seven"); break;
                case 8: System.out.println("eight"); break;
                case 9: System.out.println("nine"); break;
                default: System.out.println("ten");
          }
    }
   }
}
```

EJERCICIOS PROPUESTOS

- **1.** Realiza un programa con tres variables de tipo entero a, b y c. El programa deberá mostrar por pantalla el valor menor y mayor.
- **2.** Realiza un programa utilizando bucles que muestre la siguiente figura por pantalla:

```
*
**
***
****
*****
```

- **3.** (Ejercicio de dificultad alta) Realiza un programa utilizando bucles que muestre la siguiente figura por pantalla:

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

- **4.** (Ejercicio de dificultad alta) Realiza un programa utilizando bucles que muestre la siguiente figura por pantalla:

```
   *
  * *
 *   *
*     *
 *   *
  * *
   *
```

- **5.** (Ejercicio de dificultad alta) Se desea conocer el *lucky number* (número de la suerte) de cualquier persona. El número de la suerte se consigue reduciendo la fecha de nacimiento a un número de un solo dígito. Por ejemplo, la fecha de nacimiento de Emma es la siguiente: 16-08-1973 → 16+8+1973 = 1997 → 1+9+9+7=26 → 2+6 =8. El número de la suerte de Emma será el 8.

- Realiza un programa que calcule el *lucky number* de cualquier persona.

6. Realiza un programa que muestre por pantalla las tablas de multiplicar del 1 al 10 con el siguiente formato:

Tabla del 1
****
1 x 1 = 1
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
.....

7. Tenemos la siguiente clase:

```
import java.util.*;
public class Test {
   public static void main(String[] args) {
          Random rnd = new Random();
          int valor = rnd.nextInt();
   }
}
```

- Modifica el programa para que *valor* esté entre el rango [100-200] y muestre por pantalla si *valor* es par o impar.

8. Realiza una clase con un método *decimalToRomano* que transforme números en formato decimal a números en formato romano.

9. (Ejercicio de dificultad alta) Realiza una clase *minumero* con un *método esOmirp* que diga si un número es Omirp o no. Un número es Omirp si es un número primo y, además, al invertir sus dígitos da otro número primo. Por ejemplo: 7951 y 1597.

10. Realiza una clase *minumero* con un método *esVampiro* que diga si un número es vampiro o no. Un número es vampiro si es obtenido a partir del producto de dos números que se obtienen a partir de los dígitos del mismo (los dos colmillos). Por ejemplo: 2187=27x81 ó 1260=21x60.

11. Realiza un programa que muestre por pantalla los 50 primeros números pares.

12. Realiza un programa que muestre por pantalla los números del 1 al 100 sin mostrar aquellos números múltiplos de 5.

4 Programación orientada a objetos. Clases

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender el concepto de recursividad y saber aplicarlo en la resolución de problemas.
- ✓ Agrupar los programas y clases generadas en paquetes para crear una estructura más lógica y útil.
- ✓ Trabajar en profundidad con el concepto de clase.
- ✓ Diseñar e implementar la estructura y miembros de una clase.
- ✓ Estudiar y comprender el concepto de constructor y finalizador.
- ✓ Aplicar el concepto de herencia en la resolución de problemas.
- ✓ Comprender el concepto de interface y su aplicación en Java.

4.1 CREACIÓN DE PAQUETES

El concepto de paquete y CLASSPATH se ha visto en el capítulo 2. Sobre los paquetes hay que tener en cuenta los siguientes conceptos:

- 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 de 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:

La idea 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 4.1. Estructura de directorios de Geany

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 a crear 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.

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();
    }
}
```

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.

4.2 CONCEPTO DE CLASE

En la programación orientada a objetos las clases permiten a los programadores abstraer el problema a resolver ocultando los datos y la manera en la que estos se manejan para llegar a la solución (se oculta la implementación). 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 de 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 que haya que modificar el código sin modificar el interfaz con otras clases o programas, esto debería poder hacerse sin tener 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 publico).
- **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:

Tabla 4.1. Modificadores de acceso en Java

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.

4.2.1 CONTROL DE ACCESO A UNA CLASE

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.

Recuerda
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.

4.2.2 REFERENCIA AL OBJETO THIS

Java, al igual que C++, proporciona una referencia al objeto con el que se está trabajando. Esta referencia se denomina **this**, que 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;
    }
}
```

4.2.3 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:

Tabla 4.2. Métodos de la clase Object

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.

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 sobrescribirá 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.

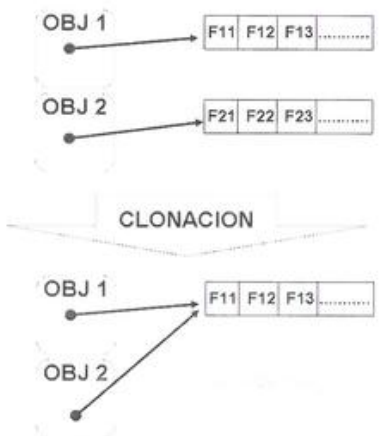


Figura 4.2. Copia superficial

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.

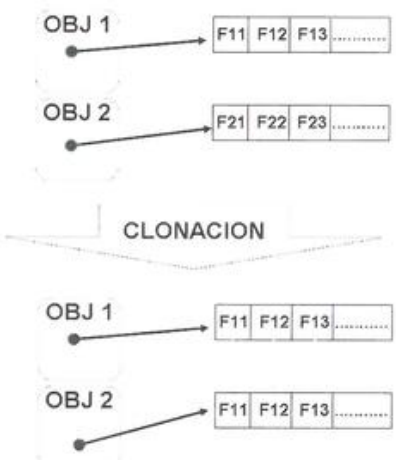


Figura 4.3. Copia en profundidad

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 de 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.

4.3 ESTRUCTURA Y MIEMBROS DE UNA CLASE

En esta sección se va a trabajar en profundidad con 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.

4.3.1 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 de 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 como 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*.

4.3.2 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.

4.3.3 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 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();
```

4.3.4 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 éste método no es estático.

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

Tabla resumen:

Tabla 4.3. Tabla resumen

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

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 son 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:

Tabla 4.4. Métodos de la clase Math

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 ó 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 ó 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 (éste último no incluido).
static int round(float a) static long round(double a)	Redondeo. Redondea el parámetro a al valor entero más cercano.

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

4.4 TRABAJANDO CON MÉTODOS

4.4.1 PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

Generalmente 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 éste los modifica. En ese caso con *return* solamente podríamos obtener una variable modificada. Por lo tanto, deberemos utilizar paso de parámetros por referencia.

Resumiendo:

- 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 explicado 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 más adelante en profundidad en el **capítulo 6**.

Gráficamente el comportamiento del programa es el siguiente:

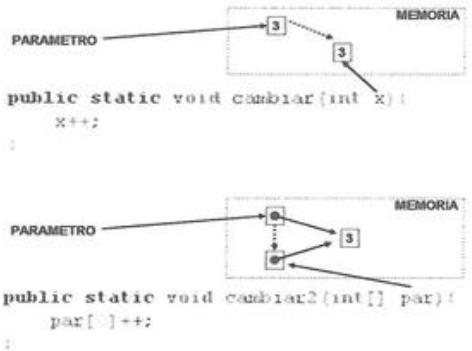


Figura 4.4. Parámetros por valor y por referencia

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.

4.4.2 LOS MÉTODOS RECURSIVOS

A FONDO

LOS MÉTODOS RECURSIVOS

Un método se llama **recursivo** cuando **se llama a sí mismo**.

¿Cuándo utilizar la recursividad?

- Cuando la resolución de un problema es más sencilla.
- Cuando no es infinita, es decir, hay un caso resoluble más básico o más sencillo.

Generalmente, cuando se va a resolver un problema recursivo vemos que en cada llamada sucesiva al método recursivo nos vamos acercando cada vez más a la solución.

¿Es eficiente la recursividad?

NO. La recursividad **no** es eficiente pero es sencilla de programar y de entender. Hay que tener siempre en cuenta que para un método recursivo siempre hay uno equivalente iterativo.

Ejemplo de recursividad

Vamos a ver la recursividad con un ejemplo sencillo. El método que vamos a escoger es la potencia de

un número. Nuestro método será el siguiente:

potencia(x,y) → xy

Para resolver un caso recursivo generalmente debemos encontrar:

1. Una fórmula o proceso que reduzca la complejidad y nos vaya acercando a la solución.
2. Un caso base que hace que nuestra recursividad no sea infinita.

En el caso de la potencia:

- Sabemos que $x^y = x * x^{y-1}$(ésta es la fórmula que reduce la complejidad).
- Y que $x^0 = 1$ (caso base).

Este problema, como se puede observar, es un claro caso de método recursivo.

La programación del método sería la siguiente:

```
public static int potencia(int x, int y){
   if (y == 1){ //caso base
   return x;
   }else{ //reducción de la complejidad
   return x * potencia(x,y-1);
   }
   }
```

Veamos como funciona una llamada al método cuando queremos realizar la operación 2^3.

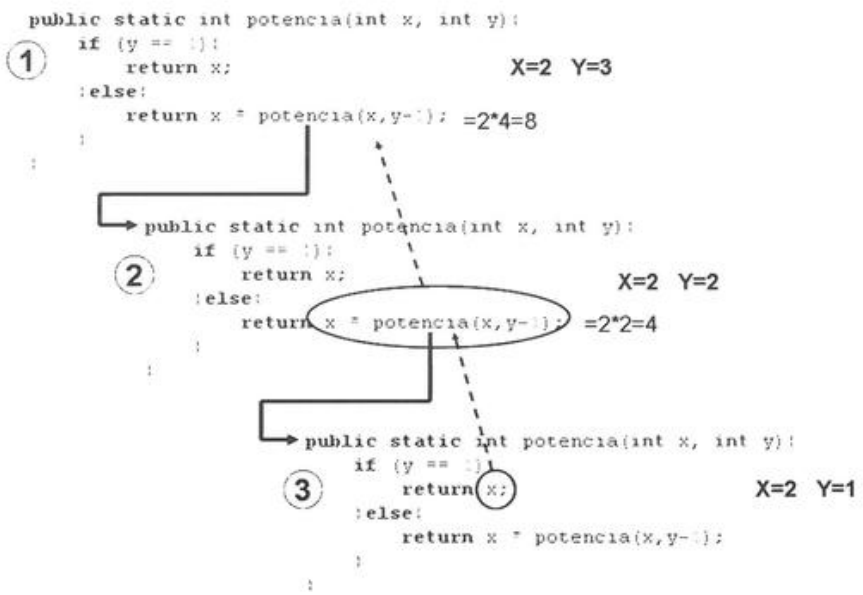


Figura 4.5. Estructura de la llamada a un método recursivo

Como se puede observar, cuando se realiza una llamada al método potencia(2,3), éste va a estar llamandose recursivamente hasta que el segundo parámetro(y) valga 1. Luego, en nuestra llamada al método se producirán dos subllamadas (potencia(2,2) y potencia(2,1)). En la última llamada (cuando y = 1) se produce el fin de las llamadas recursivas y se procede a recuperar los valores obtenidos en las subllamadas (retorno de los valores 2 y 4). Al final, el método devuelve 8 (2 * el valor retornado que es 4).

Para entender más sobre la recursividad se recomienda trabajar en profundidad los ejercicios resueltos.

4.5 LOS CONSTRUCTORES

Java, al igual que hace con las variables, cuando va a crear un objeto, lo que hace es reservar espacio en 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 en la sección a fondo al 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.

Tabla 4.5. Inicialización predeterminada de variables

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

Figura 4.6. Pasos enla creación de los objetos

4.5.1 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);
```

4.5.2 ASIGNACION 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. De forma sencilla 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 Ancho: 5	Alto: 8 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:

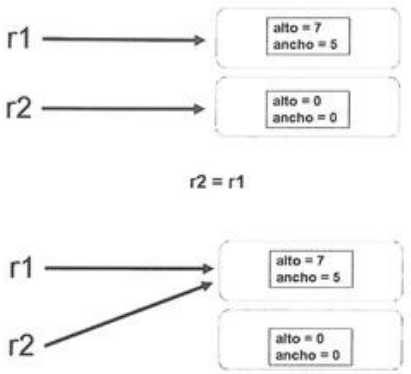


Figura 4.7. Asignación de referencias

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

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

4.5.3 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 clase anterior 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

A FONDO

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.
- Ideal 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 utilizados en las clases anónimas y no tienen el modificador *static*.

Un ejemplo muy sencillo de clase 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...
```

4.6 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 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).

4.6.1 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 ejecutarlos finalizadotes 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.

Importante
El *garbage collector*, gc o recolector de basura, se ejecuta en segundo plano en un subproceso paralelo a la propia aplicación. La llamada al recolector de basura se hace ejecutando el método *gc()* de la clase *System*.

Resumiendo:

- 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 ésta 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.

4.7 ENCAPSULACIÓN Y VISIBILIDAD. 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 es un grupo de métodos con sus cuerpos vacíos. Por ejemplo, la interfaz figura (intfigura) podría ser el 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, ésta debe contener los métodos declarados en dicha interfaz.

4.8 HERENCIA

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

Recuerda
En Java al contrario que en C++ no se permite la herencia múltiple. Es decir, una clase no puede heredar de varias clases.

En la siguiente figura se muestra un ejemplo de herencia:

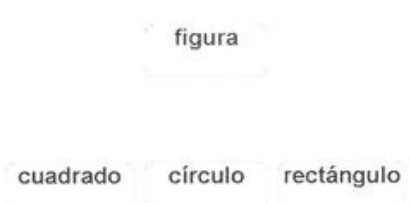


Figura 4.8. Estructura de clases descendientes de figura

Como se puede ver en la figura, en este árbol de herencia tendremos la clase *figura* de las 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 de 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 ésta no hereda de ninguna clase concreta en realidad hereda 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 la clase *figura* y *cuadrado* se muestra junto a la siguiente imagen:

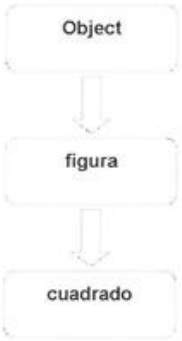


Figura 4.9. Jerarquía de clases

```
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** lo que 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*.

RESUMEN DEL CAPÍTULO

En este capítulo se profundiza y se amplían los conceptos que se vieron en el Capítulo 2. Una vez estudiado este capítulo el alumno se dará cuenta que en el Capítulo 2 se vieron solamente los conceptos básicos para poder realizar nuestros primeros programas. En este capítulo se entrará a estudiar en profundidad el concepto de clase así como los miembros y estructura de una clase. Se verán los métodos recursivos. El alumno deberá de comprender a fondo el concepto de recursividad para poder realizar los ejercicios propuestos. También se verán en este capítulo algunas de las características más importantes de Java, como son las interfaces y la herencia. Se recomienda al alumno un estudio exhaustivo de todos los apartados para luego poder profundizar más a lo largo del siguiente tema.

EJERCICIOS RESUELTOS

1A. Realiza una clase con un método factorial que utilizando la recursividad genere el factorial de un número dado.

Solución:

Como se dijo antes en el desarrollo del capítulo, para resolver un caso recursivo debemos encontrar:

a. Una fórmula o proceso que reduzca la complejidad y nos vaya acercando a la solución.

En nuestro caso sabemos que por ejemplo factorial(4) es 4 * factorial(3), lo que es igual factorial(num) = num * factorial(num -1).

b. Un caso base que hace que nuestra recursividad no sea infinita.

En nuestro caso será factorial(0) = 1.

En la siguiente figura se muestra el factorial de una manera más matemática:

$$n! \quad \begin{cases} \text{Si } n = 0 \rightarrow 1 \\ \text{Si } n >= 1 \rightarrow (n-1)! \cdot n \end{cases}$$

Figura 4.10. Algoritmo factorial

El código que resuelve el factorial es el siguiente:

```
class Test {
    public static int factorial(int num)
    {
          if (num == 0) return 1;
          return num * factorial(num-1);
    }
    public static void main(String[] args) {
          System.out.println("El factorial de 0 es : "+factorial(0));
          System.out.println("El factorial de 1 es : "+factorial(1));
          System.out.println("El factorial de 2 es : "+factorial(2));
          System.out.println("El factorial de 3 es : "+factorial(3));
          System.out.println("El factorial de 4 es : "+factorial(4));
          System.out.println("El factorial de 5 es : "+factorial(5));
    }
}
```

1B. Para el programa anterior, modifica el método para generar el factorial de forma iterativa.

Solución:

```
class Test {
    public static int factorial(int num)
    {
          int factorial=1;
          while(num>0){

                factorial *= num;
                num--;
          }
          return factorial;
    }
    public static void main(String[] args) {
          System.out.println("El factorial de 0 es : "+factorial(0));
          System.out.println("El factorial de 1 es : "+factorial(1));
          System.out.println("El factorial de 2 es : "+factorial(2));
          System.out.println("El factorial de 3 es : "+factorial(3));
          System.out.println("El factorial de 4 es : "+factorial(4));
          System.out.println("El factorial de 5 es : "+factorial(5));
    }
}
```

¿Qué ventaja tiene la solución iterativa frente a la recursiva?

La solución iterativa tiene la ventaja de la rapidez de ejecución (más eficiente). La solución recursiva es más lenta pero en muchas ocasiones (en esta concretamente no) es mucho más fácil de entender y más fácil de codificar con lo cual, aunque tengamos un código menos eficiente es mucho más fácil de comprender y mantener.

Como ejercicio complementario modifica los métodos anteriores para que cuando se introduzca un número menor a 0 el método muestre un mensaje de error y no realice ningún cálculo.

2. Realiza un programa con un método recursivo que muestre por pantalla la siguiente serie:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Solución:

Esta serie es un problema ampliamente conocido como serie de Fibonacci. Se caracteriza por lo siguiente:

Fibonacci(0) = 0

Fibonacci(1) = 1

Y para todos los demás números:

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

El código que resuelve la serie es el siguiente:

```
class Test {
    public static int fibonacci(int num)
    {
          if (num == 0) return 0;
          if (num == 1) return 1;
          return fibonacci(num-1)+fibonacci(num-2);
    }
    public static void main(String[] args) {
          for (int i=0;i<10;i++){
                System.out.print(fibonacci(i)+", ");
          }
    }
}
```

3. Realiza un programa que utilizando recursividad muestre por pantalla la siguiente pirámide:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

Figura 4.11. Triángulo numérico

El programa podrá generar una pirámide de cualquier número de filas.

Solución:

¿Cómo resolver el problema?

El primer paso, desde mi punto de vista, es observar el problema desde otra perspectiva. El mirar el problema como una pirámide evita poder resolverlo de forma satisfactoria. Si modificamos el programa e intentamos resolverlo como si fuese una pirámide tendría más sentido. La pirámide anterior en forma matricial sería la siguiente:

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

De esta matriz podemos concluir lo siguiente:

- Los elementos con valor 0 no se mostrarán.
- El primer elemento de cada fila vale 1.
- Los elementos cuya columna sea menor de 1 valen 0.
- Los elementos cuya columna sea mayor que la fila valen 0.
- Para los demás elementos, el elemento(fila, columna) es igual al elemento(fila-1,columna)+elemento(fila-1,columna-1)

Una vez que tenemos claros estos principios el siguiente paso es programar el método. Observa que es la última regla la que implica que este método sea recursivo.

Con esto ya tenemos casi realizado el programa. La programación de los bucles no parece compleja a estas alturas pero lo que puede resultar más complejo es "enderezar" la pirámide introduciendo espacios en blanco antes de imprimir cualquier número. En nuestro caso se ha resuelto el problema añadiendo numfilas-filaactual o mejor dicho numfilas-i espacios en blanco en cada línea y de esta manera se imprimirán 4, 3, 2, 1 y 0 espacios en blanco en cada línea.

```
class piramide {
    public static int elemento(int fila, int columna){
           if (columna == 1 ) return 1;
           if (columna < 1 || columna > fila) return 0;
           return elemento(fila-1,columna)+elemento(fila-1,columna-1);
    }
    public static void main(String[] args) {
           int numfilas = 5;
           for (int i=1; i<(numfilas+1) ; i++){
                  for (int e=0; e<(numfilas - i);e++)System.out.print(" ");
                  for (int j=1; j<(numfilas+1) ; j++){
                         int dato = elemento(i,j);
                         if (dato > 0 ) System.out.print(dato+" ");
```

```
                }
                System.out.println("");
            }
        }
    }
```

Este programa funcionará con una pirámide de cualquier número de filas, bastará solamente con cambiar el valor a la variable numfilas por el número de filas deseado. Nótese que cuando se comienzan a mostrar números de dos y más cifras la alineación de la pirámide se pierde. Se pide al alumno como ejercicio complementario que solvente este problema.

4. Realiza un programa que utilizando la recursividad muestre por pantalla la siguiente pirámide:

```
        1
      1 1 1
    1 2 3 2 1
  1 3 6 7 6 3 1
```

Figura 4.12. Triángulo numérico (II)

El programa podrá generar una pirámide de cualquier número de filas.

Solución:

```
class piramide {
    public static int elemento(int fila, int columna){
       if (fila < 1 || columna < 1 ) return 0;
       if (columna == 1 ) return 1;
return elemento(fila-1,columna)+elemento(fila-1,columna-1)+elemento(fila-1,columna-2);
    }
    public static void main(String[] args) {
           int numfilas = 4;

           for (int i=1; i<(numfilas+1) ; i++){
                 for (int e=0; e<(numfilas - i);e++)System.out.print("  ");
                 for (int j=1; j<(2*numfilas+1) ; j++){
                       int dato = elemento(i,j);
                       if (dato > 0 ) System.out.print(dato+" ");
                 }
                 System.out.println("");
           }
    }
}
```

Este programa funcionará con una pirámide de cualquier número de filas, bastará solamente con cambiar el valor a la variable numfilas por el número de filas deseado. Nótese que cuando se comienza a mostrar números de dos o más cifras la alineación de la pirámide se pierde. Se pide al alumno como ejercicio complementario que solvente este problema.

5. Realiza una clase TransformaBase la cual trasforme y cambie de base números decimales.

Solución:

```
class TransformaBase {
    public static void muestraCifra(int dat) {
        if (dat<10) {
            System.out.print(dat);
        } else {
    dat-=10;
    char c = (char)('A'+ dat);
            System.out.print(c);
        }
    }
    public static void transforma(int dato, int base) {
        if (base > dato) {
            muestraCifra(dato);
        } else {
            transforma(dato/base,base);
            muestraCifra(dato%base);
        }
    }
   public static void main(String[] args) {
        transforma(8,2);
        System.out.println("");
        transforma(12,16);
        System.out.println("");
        transforma(13,8);
        System.out.println("");
   }
}
```

Como ejercicio complementario se pide al alumno que cree otro método en el cual se transformen los números de forma iterativa.

6. En la siguiente clase indica cuál es el atributo de instancia y cuál es el atributo de clase:

```
public class unaClase {
    public static int a = 20;
    public int b = 13;
}
```

Solución:

- La variable de clase es la a.
- La variable de instancia es la b.

EJERCICIOS PROPUESTOS

- **1.** Realiza un programa que muestre por pantalla el siguiente cuadrado:

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

Figura 4.13. Matriz numérica

El programa podrá generar un cuadrado de cualquier dimensión. Utiliza la recursividad para resolver el problema.

- **2.** (Ejercicio de dificultad alta) Realiza los ejercicios resueltos de forma recursiva de tal forma que la solución ahora sea iterativa.

- **3.** (Ejercicio de dificultad alta) Para los ejercicios resueltos de pirámides crea como añadido un método recursivo que muestre la suma de los valores mostrados por pantalla.

- **4.** Crea en tu equipo un paquete Utilidades.mates con dos clases sumar y potenciar. La clase sumar tendrá un método int suma(int,int) el cual devolverá la suma de los dos parámetros introducidos y la clase potenciar tendrá un método int potencia(int,int) el cual devolverá el resultado de elevar el primer parámetro al segundo parámetro. Realiza un programa que haga uso de este paquete.

- **5.** Realiza una clase *pez* la cual tendrá un miembro nombre de tipo *String* el cual podrá ser heredado por sus subclases. Realiza un método getNombre y otro setNombre. Utiliza el objeto *this* en estos métodos. Implementa en esta clase el método *clone()* así como el método *equals()* para poder hacer una comparación en profundidad. Realiza un programa que haga un testeo en profundidad de las características de esta clase.

- **6.** Para la clase *pez* anterior, crea un miembro privado entero numpeces común a todos los objetos *pez* el cual cuente el número de peces creados. Crea un programa que compruebe que esta variable se incrementa cada vez que se crea un objeto *pez*.

7. Para el objeto *pez* anterior crea un constructor copia. Comprueba este constructor mediante un programa.

8. Crea una clase *prueba* en el que tenga dos métodos (*primero* y *segundo*). El método *segundo* llamará al método *primero* dos veces, de forma normal y utilizando *this*. Verifica que ambas llamadas son equivalentes.

9. ¿Qué mostrará el siguiente programa por pantalla?

```
public class bebe {
    bebe (int i) {
          this("Soy un bebe consentido");
          System.out.println("Hola, tengo " + i + " meses");
    }
    bebe (String s) {
          System.out.println( s );
    }
    void berrea() {
          System.out.println("Buaaaaaaaaaa");
    }
   public static void main(String[] args) {
                new bebe (8).berrea();
   }
}
```

5 P.O.O. Utilización avanzada de clases

OBJETIVOS DEL CAPÍTULO

- ✓ Seguir profundizando en los conceptos de Orientación a Objetos.
- ✓ Trabajar con clases de utilidad como *wrappers* y clases que manejan fecha y hora.
- ✓ Aprender nuevos conceptos como clases y métodos abstractos.
- ✓ Mejorar y ampliar las posibilidades del diseño y programación de aplicaciones utilizando conceptos como polimorfismo, sobrescritura, *overloading*, *casting*, etc.
- ✓ Comprender el concepto de clases anidadas.

5.1 WRAPPERS

La principal diferencia entre un tipo primitivo y un **wrapper** es que éste último **es una clase**. Cuando trabajamos con *wrappers* estamos trabajando con objetos mientras que cuando trabajamos con un tipo primitivo obviamente no.

A simple vista un *wrapper* siendo un objeto, puede aportar muchas ventajas pero, ¿qué problemas nos puede plantear utilizar objetos en vez de tipos primitivos?

El problema que nos podemos encontrar es que cuando le pasamos una variable a un método como argumento y esta es de un tipo primitivo se le pasa siempre por valor mientras que cuando pasamos un *wrapper* (objeto) se lo estamos pasando por referencia.

Consejo
Echa un vistazo al apartado del tema 4 en el que se explica el paso de parámetros por valor y referencia.

Una de las grandes ventajas que tienen estos *wrapper* es la facilidad de conversión entre tipos primitivos y cadenas de caracteres en ambos sentidos. Existen *wrappers* de todos los tipos primitivos numéricos (*Byte*, *Short*, *Integer*, *Double*, etc).

Tabla 5.1. Tipos primitivos y wrappers asociados

Tipo primitivo	Wrapper asociado
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

5.1.1 CLASE WRAPPER INTEGER

El *wrapper Integer* tiene dos constructores:

- Integer(int)
- Integer(String)

En la siguiente tabla se muestra un resumen de algunos de los métodos del *wrapper Integer*:

Tabla 5.2. Métodos del wrapper Integer

Método	Descripción
Integer(int) Integer(String)	Constructores.
byteValue() shortValue() intValue() longValue() doubleValue() floatValue()	Funciones de conversión con datos primitivos.
Integer decode(String) Integer parseInt(String) Integer parseInt(String,int) Integer valueOf(String) String toString()	Conversión a *String*.
String toBinaryString(int) String toHexString(int) String toOctalString(int)	Conversion a otros sistemas de numeración.
MAX_VALUE, MIN_VALUE, TYPE	Constantes.

La creación de un objeto *Integer* es la misma que para cualquier otro tipo de objeto:

```
Integer i2 = new Integer("7");
Integer i1 = new Integer(5);
```

Recuerda
Una vez asignado un valor al objeto *Integer* no puede cambiarse. Si queremos utilizar otro valor deberemos crear otro objeto *Integer*

En el siguiente programa se muestra la utilización de algunos de los métodos expuestos en la tabla anterior:

```
class test {
   public static void main(String[] args) {
    Integer i1 = new Integer(5);
    Integer i2 = new Integer("7");
    String s1 = i1.toString();
    System.out.println(s1);//muestra 5 por pantalla
    int i3 = Integer.parseInt("10",10);
    int i4 = Integer.parseInt("10",8);
    int i5 = Integer.parseInt("BABA",16);
    System.out.println(i3);//muestra 10 por pantalla
    System.out.println(i4);//muestra 8 por pantalla
    System.out.println(i5);//muestra 47.802 por pantalla
    System.out.println(Integer.toOctalString(i4));//muestra 10 por pantalla
    System.out.println(Integer.toHexString(i5));//muestra baba por pantalla
    int i6 = Integer.valueOf("22").intValue();
    System.out.println(i6);//muestra 22 por pantalla
    }
}
```

Prácticamente la totalidad del código mostrado anteriormente es fácilmente interpretable por el alumno, pasaremos a comentar solamente algunas de estas líneas:

```
int i3 = Integer.parseInt("10",10);
int i4 = Integer.parseInt("10",8);
int i5 = Integer.parseInt("BABA",16);
```

En las anteriores líneas se utiliza el método parseInt(Strings, int base), el cual permite introducir la base en la que está codificado el número del primer parámetro.

```
System.out.println(i5);  // muestra 47.802 por pantalla
```

En esta línea muestra el número 47.802 que equivale en decimal al número BABA en hexadecimal. Para mostrar dicho número en su base correspondiente hay que hacer uso del método *toHexString()*.

```
int i6 = Integer.valueOf("22").intValue();
```

En la línea de código anterior se puede ver como el objeto *Integer* hace una conversión de un *String* (*valueOf*) almacenando dicho valor en una variable miembro interna para luego devolverla mediante el método *intValue()*.

Recuerda
Los wrappers para los demás tipos primitivos tienen una funcionalidad y modo de utilización similar al *wrapper Integer*.

5.2 TRABAJANDO CON FECHAS Y HORAS (LA CLASE DATE)

La clase *Date* es una clase de utilidad contenida en el *package* java.util. Con la clase *Date* podemos representar un instante dado con precisión de milisegundos. La fecha y hora se almacenan en un entero de tipo *Long* que registra los milisegundos transcurridos desde el 1 de enero de 1970 GMT (Tiempo del meridiano de Greenwich) a las 00:00:00.

Recuerda
Existen otras clases que permiten obtener información del año, mes, día, hora, minutos y segundos de un objeto *Date*.

Para la utilización de fechas se suele trabajar con otro tipo de clases, como la clase *GregorianCalendar*, la cual deriva de la clase abstracta *Calendar*. Al ser *Calendar* una clase abstracta, cuando queramos utilizar variables de tipo fecha en nuestro programa lo haremos a través de objetos de la clase *GregorianCalendar*.

En la clase *GregorianCalendar* las horas se representan por un número entre 0 y 23, los días entre 1 y 31, los años se representan con cuatro dígitos. El problema radica en que los meses del año se representan con un entero que va de 0 a 11. En el ejemplo siguiente se puede ver como se añade 1 al mes puesto que enero es el 0 y diciembre el 11. Esta clase tiene muchas variables enteras entre otras las siguientes: DAY_OF_WEEK, DAY_OF_MONTH, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND, WEEK_OF_MONTH, WEEK_OF_YEAR, etc.

En el siguiente código se muestra un ejemplo de utilización de la clase *Date*:

```
import java.util.*;
class fecha {
    public static void main(String[] args) {
           Date d = new Date();
           GregorianCalendar c = new GregorianCalendar();
           c.setTime(d);
```

```
            System.out.print(c.get(Calendar.DAY_OF_MONTH));
            System.out.print("-");
            System.out.print(c.get(Calendar.MONTH)+1);
            System.out.print("-");
            System.out.println(c.get(Calendar.YEAR));
    }
}
```

El programa anterior muestra por pantalla la fecha actual.

5.3 CLASES Y MÉTODOS ABSTRACTOS Y FINALES

La abstracción es una de las características principales de la programación orientada a objetos. Mediante la abstracción lo que se hace es extraer la esencia básica y su comportamiento para luego después representarla en un lenguaje de programación.

Recuerda
Abstract en Java es sinónimo de genérico.

5.3.1 CLASES Y MÉTODOS ABSTRACTOS

Las clases abstractas son clases que son y han sido pensadas para ser genéricas. Esto quiere decir que no va a haber objetos de esas clases puesto que no tiene sentido. Por ejemplo, la clase vehículo es claramente una clase genérica porque cuando implemente un programa con esta clase no voy a crear vehículos sino objetos de la clase *coche* u objetos de la clase *moto*, etc. Es obvio que todos son vehículos y por lo tanto esta clase abstracta solamente definirá los atributos y comportamientos (métodos) comunes. Ejemplos de dichos atributos pueden ser color, peso, etc. y ejemplos de estos métodos pueden ser *getVelocidadActual()*. Obviamente, la velocidad se calculará de manera diferente para cada tipo de vehículo.

Un ejemplo de la clase abstracta vehículo podría ser el siguiente:

```
public abstract class vehiculo{
    private int peso;
    public void setPeso(int p){peso=p;}
    public abstract int getVelocidadActual();
}
```

Como puedes observar en el ejemplo anterior, una clase abstracta puede implementar métodos abstractos y no abstractos.

Recuerda

- De las clases abstractas no pueden crearse objetos.
- Si una clase tiene métodos abstract por fuerza tendrá que ser una clase abstracta.
- Un método *abstract* no puede ser *static*.
- Las subclases que implementen esta clase abstracta tendrán que redefinir estos métodos o bien declararlos también como *abstract*.

5.3.2 OBJETOS, CLASES Y MÉTODOS FINALES

Objetos finales

Cuando un objeto se declara como final, éste impedirá que haya otro objeto con la misma referencia. Por ejemplo cuando se realice algo parecido a esto:

```
final cuadrado c1=new cuadrado(5);
cuadrado c2=new cuadrado(15);
c1=c2;
```

El compilador mostrará un mensaje de error en la tercera línea. Más concretamente Geany dará el siguiente mensaje de error:

```
prueba.java:6: cannot assign a value to final variable c1
  c1=c2;
  ^
1 error
La compilación falló.
```

Métodos finales

Cuando declaramos un método como final, le estamos diciendo al compilador que ese método no va a cambiar. No va a ser sobrescrito, con lo cual el compilador puede colocar el *bytecode* del método justo en el sitio del programa donde va a ser invocado con la consiguiente **ganancia de eficiencia**.

```
public final void setColor(String s){color=s;}
```

Clases finales

Cuando una clase se declara como final, esa clase no puede tener descendencia (no puede tener subclases).

Por ejemplo, la siguiente clase triángulo no podrá tener subclases que deriven de ella:

```
public final class triangulo{
.....
}
```

5.4 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 esta es 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 5.1. Jerarquía de clases

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 ó 550?

La solución es **550**. Aunque la referencia se creo para la clase *empleado* y solamente se pueden llamar a métodos de dicha clase, el método *getSueldo()* está sobrescrito 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 más profundidad en el siguiente apartado.

Recuerda
Cuando la referencia se creo para la clase base, solamente se pueden hacer llamadas a métodos de dicha clase base.

Vamos a ver en más 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 el código que se va a ejecutar en dicha llamada), la vinculación temprana y la vinculación tardía.

- **Vinculación temprana**. Se realiza en **tiempo de compilación**. Con métodos normales o sobrecargados Java utiliza la vinculación temprana.
- **Vinculación tardía**. Se realiza 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 sobrescrito 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 concebido 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 y por lo tanto, de momento se incluyen las líneas que dan problema y debajo la línea reprogramada solventando 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
```

5.5 SOBREESCRITURA 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 sobrescribir; con los elementos miembro esta técnica no es posible.

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

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

- Tiene que tener el mismo nombre (esto es obvio).
- El retorno de la clase padre e hijo deberá de ser del mismo tipo.
- Deberá de 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* sobrescribe 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*).

5.6 SOBRECARGA DE MÉTODOS (OVERLOADING)

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 de 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 de los que por costumbre no se utiliza 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 de cambiar la lista de argumentos del método.

5.7 CONVERSIONES ENTRE OBJETOS (CASTING)

En capítulos anteriores hemos visto la esencia del *casting* cuando convertíamos un tipo primitivo en otro (generalmente con más precisión). El *casting* es la conversión de unos tipos u objetos en otros. Como hemos visto, para convertir un objeto en otro debe de haber entre ambos objetos una relación de herencia (uno debe de ser una subclase del otro). Dado que la subclase contiene toda la información que pueda contener su superclase es lógico pensar que el *casting* es posible. En este caso no es necesario hacer *casting* dada la correspondencia de información entre una clase y otra.

En el siguiente código se muestra un ejemplo de *casting* de un tipo primitivo (conversión de *integer* a *long*):

```
int i = 50;
long l;
l=(long)i;
```

Imaginemos el siguiente árbol de jerarquía:

Figura 5.2. Jerarquía de clases

Si por ejemplo tenemos un método *m()* que espera un argumento tipo *empleado*, podemos pasarle un objeto de tipo *persona* **(1)** o un objeto de tipo *encargado* **(2)**.

Opción 1. Si le pasamos un objeto de tipo *persona* nos encontramos con una pérdida de precisión puesto que no se pueden ejecutar todos los métodos de los que dispone un objeto de tipo *empleado*. Recuerda que *persona* contiene menos métodos que la clase *empleado*.

En este caso es necesario hacer un *casting*, sino el compilador dará error.

Imaginemos el siguiente código:

```
    public static void m(empleado e){
          System.out.println(e.getNombre());
    }
public static void main(String[] args) {
          persona p1 = new empleado();
          p1.setNombre("Isaac Sanchez");
          encargado en1 = new encargado();
          en1.setNombre("Andrés Rosique");
          m(en1);
          //m(p1); //esta llamada sin casting dará error de compilación
          m((empleado)p1);
          persona p2 = new persona();
          p2.setNombre("Juan Serrano");
          //m((empleado)p2); //esta llamada dará error en ejecución
    }
```

Como se puede observar tenemos un método *m()* que espera como parámetro un objeto de tipo *empleado*. En primer lugar haremos que la referencia p1 a persona apunte al objeto de tipo *empleado()*. Una vez hecho esto no tendremos problemas para utilizar esta referencia en el método. Para ejecutar el método tendremos que hacer **explícitamente un casting** como se puede ver en la siguiente línea de código:

```
m((empleado)p1);
```

Este *casting* es similar a lo que hemos visto antes con los métodos primitivos.

Si la referencia p1 en vez de apuntar a un objeto de tipo *empleado* apuntase a un objeto de tipo persona. En ese caso utilizando siguiente línea de código:

```
persona p1 = new persona();
```

Java dará un error en ejecución. Java lanzará una excepción del tipo ClassCastException y avisará que no es posible realizar el *casting* "persona cannot be cast to empleado".

Opción 2. Pasamos un objeto de tipo *encargado*. En este caso al ser una subclase no tendremos problemas.

Recuerda
Los errores se producen cuando se llama a métodos que el objeto destino no tiene.

Resumiendo:

```
empleado emp = new empleado();
encargado enc = new encargado();
emp = enc; //No necesita casting
enc = (encargado)emp; //necesita casting explícito
```

Las reglas seguidas en el ejemplo anterior son las siguientes:

- Cuando se utiliza una clase **más específica** (más abajo en la jerarquía) **no** hace falta *casting*.
- Cuando se utiliza una clase **menos específica** (más arriba en la jerarquía) hay que hacer un *casting* explícitamente.

5.8 ACCESO A MÉTODOS DE LA SUPERCLASE

Para acceder a los métodos de la superclase se utilizará la palabra reservada **super**. Esta palabra reservada está disponible en cualquier método no estático de una subclase.

Es importante tener en cuenta que *super* es una referencia al objeto actual teniendo en cuenta la instancia de su superclase.

Echemos un vistazo al siguiente código:

```
class padre{
    protected int dato;
    public void m(){
          System.out.println("método clase padre");
    }
}

class hijo extends padre{
    private int dato;
    public void m(){
          System.out.println("método clase hijo");
          super.dato = 10;
          dato = 20;
    }
    public void getDato(){
          System.out.println(super.dato);
    }
    public void mostrar(){
          this.m();
          m();
          super.m();
    }
}

class test {
    public static void main(String[] args) {
          hijo h = new hijo();
          h.mostrar();
          h.getDato();
    }
}
```

El resultado en pantalla de ejecutar este código será el siguiente:

```
método clase hijo
método clase hijo
método clase padre
10
Presione una tecla para continuar . . .
```

De este código extraemos lo siguiente:

- Para acceder a métodos sobrescritos de la superclase utilizamos la palabra *super*.
- Es posible acceder a miembros *protected* de la superclase utilizando la palabra *super*.

Recuerda

this se utiliza para acceder a campos y métodos de la clase y super para la superclase. No importa que estos estén sobrescritos.

Cambiemos un poco el código anterior y veamos cómo lo visto anteriormente con *super* y *this* es aplicable también en las llamadas a los constructores de los objetos.

```
class padre{
    protected int dato1, dato2;
    padre(int x,int y){dato1 = x; dato2 = y;}
    padre(){
          this(5,5);
    }
}

class hijo extends padre{
    private int dato1, dato2;
    hijo(int x,int y){
          super(2,2);
          dato1 = x;
          dato2 = y;
    }
    hijo(){
          dato1 = 3;
          dato2 = 3;
    }
    public void getDato(){
          System.out.println("Padre dato1:" + super.dato1);
          System.out.println("Padre dato2:" + super.dato2);
          System.out.println("hijo dato1:" + this.dato1);
          System.out.println("hijo dato2:" + this.dato2);
    }
}

class test {
    public static void main(String[] args) {
          hijo h1 = new hijo(1,1);
          h1.getDato();
          hijo h2 = new hijo();
          h2.getDato();
    }
}
```

El código anterior mostará por pantalla lo siguiente:

```
Padre dato1:2
Padre dato2:2
hijo dato1:1
hijo dato2:1
Padre dato1:5
Padre dato2:5
hijo dato1:3
hijo dato2:3
Presione una tecla para continuar . . .
```

Obsérvese como se utiliza **this** y **super** en las llamadas a los constructores.

A FONDO

MÁS UPCASTING, DOWNCASTING Y UTILIZACIÓN DE MÉTODOS AVANZADOS

Tenemos la clase abstracta **forma** la cual es antecesora de la clase hijo **círculo**. En el siguiente código está implementada esta relación jerárquica. Echa un vistazo al código y pon atención a lo siguiente:

- Observa cómo se hace el *upcasting* y *downcasting* en el método *main*.
- Observa cómo se utiliza el operador **instanceof**.
- Detente a observar y comprender el método jerarquía de la clase círculo y fíjate en la utilización de los métodos **getClass**(), **getSuperclass**() y **newInstance**().

El método jerarquía va a intentar mostrar el árbol genealógico de la clase *círculo* pero como la clase *forma* es abstracta (*abstract*), ésta no podrá ser instanciada y lanzará una excepción al intentar crear una instancia suya.

```
abstract class forma {
        void identidad() { System.out.println(this); }
        abstract public String toString();
}
class circulo extends forma {
        public String toString() { return "círculo"; }
        public static void jerarquia(Object obj) {
        Object o = obj;
        while (o.getClass().getSuperclass()!= null) {
        try {
        System.out.println(o.getClass() + " es una subclase de "+o.getClass().
getSuperclass());
        o = o.getClass().getSuperclass().newInstance();
        } catch(InstantiationException e) {
```

```
        System.out.println("Imposible instanciar la clase "+o.getClass().
    getSuperclass());
        break;
        } catch(IllegalAccessException e) {
        System.out.println("No hay acceso");
        break;


        }
        public static void main(String[] args) {
        circulo c = new circulo();
        // Haciendo el Upcast
        forma f = (forma)c;
        f.identidad();
        // haciendo el downcast
        if(f instanceof circulo)
        ((circulo)f).identidad();
        else if(!(f instanceof circulo))
        System.out.println("  f  (forma)  no  es  un  círculo");
        jerarquia(c);
        }
}
```

5.9 CLASES ANIDADAS

Una clase anidada es una clase que es miembro de otra clase.

Consejo
Antes de anidar una clase pregúntate antes lo siguiente: ¿Es necesario anidar la clase?

La definición de una clase anidada sería la siguiente:

```
class externa {
    ...
    class anidada {
        ...
    }
    ...
}
```

Dado que la clase anidada es un miembro de la clase externa, tendrá acceso a todos sus métodos y atributos (incluso a los privados). Y como es lógico, al ser un miembro de la clase externa, la clase anidada podrá ser *private*, *public*, *protected* o privada al paquete.

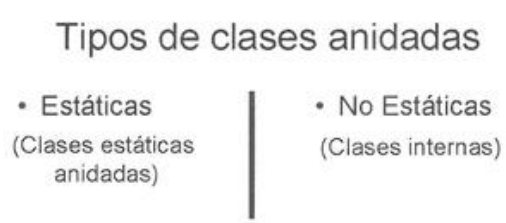


Figura 5.3. Clasificación de clases anidadas

Como se puede observar en la figura anterior, existen dos tipos de clases anidadas:

- Estáticas. También llamadas **clases estáticas anidadas**.
- No estáticas. **Clases internas**.

```
class externa{
    ...
    static class estaticaanidada {
        ...
    }
    class interna {
        ...
    }
    ...
}
```


Recuerda
Las clases estáticas anidadas se diferencian de las internas por la cláusula *static*.

Para instanciar una clase interna se seguirá el siguiente formato:

```
externa.interna objetointerno = objetoexterno.new interna();
```

De esta manera, como se puede observar, hay que instanciar primero la clase externa para luego instanciar la clase interna dentro del objeto externo.

¿Cuándo se pueden utilizar clases anidadas?

Cuando la clase solo se va a utilizar en un único lugar, en ese caso el definir la clase como anidada puede hacer que el código sea más legible y su mantenimiento sea más sencillo. También se incrementa la encapsulación dado que la clase anidada solo se necesita en la clase externa y de esta manera se mantienen juntas.

A FONDO

SISTEMAS DE OBJETOS DISTRIBUIDOS Y CORBA

Actualmente, las organizaciones y organismos funcionan de un modo distribuido. En grandes empresas, proyectos de investigación, comunicaciones, es necesario una lógica de procesamiento distribuido. Los sistemas distribuidos son componentes lógicos que se ejecutan en múltiples computadores pero de una forma integrada entre unos y otros para conseguir un objetivo común. En sistemas operativos es muy importante la estructura de comunicaciones subyacente al sistema.

CORBA es el acrónimo de *Common Object Request Broker*. En este estándar de **OMG** (*Object Management Group*) se exponen las especificaciones dirigidas a diseñadores y desarrolladores de software que permiten desarrollar aplicaciones distribuidas y heterogéneas basadas en la interoperabilidad de objetos.

CORBA es un estándar basado en una arquitectura distribuida y abierta que está basada en tres conceptos fundamentales:

- El **ORB** (*Object Request Broker*). Forma parte del núcleo de CORBA y es un gestor de objetos que se comuniquen los objetos de una manera independiente a la plataforma siguiendo una estructura cliente-servidor.
- Mecanismos para la **especificación de interfaces** como por ejemplo:
  - **IDL** (*Interface Definition Language*). Es un lenguaje puramente descriptivo utilizado para describir interfaces a los cuales los objetos cliente puedan llamar y puedan ser implementados por algún objeto.
  - **DII**(*Dynamic Invocation Interface*). Es una API (*Application Programming Interface*) utilizada en tiempo de compilación que permite la construcción dinámica de llamadas a objetos CORBA.
- Protocolos binarios para comunicación entre ORBs:
  - **GIOP**. Protocolo abstracto para la comunicación entre ORBs en la que **IIOP** es la implementación de GIOP para TCP/IP

RESUMEN DEL CAPÍTULO

Este tema es una continuación del Capítulo 4. En este tema se verán más conceptos de la Programación Orientada a Objetos como es el polimorfismo, overloading, casting, clases anidadas, etc. También se estudiarán clases de utilidad como son los wrappers y clases para el manejo de la fecha y la hora. Otros conceptos como las clases y métodos abstractos y finales también se abordan en este capítulo.

EJERCICIOS RESUELTOS

1. Crea una clase *empleado* y una subclase *encargado*. Los encargados reciben un 10% más de sueldo base que un empleado normal aunque realicen el mismo trabajo. Implementa dichas clases en el paquete sobrescribe y sobrescribe el método *getSueldo()* para ambas clases. Variables miembro no deberán de poder ser accesibles desde el exterior.

Solución:

```
package sobreescribe;

public class Empleado {
protected int sueldobase;
public int getSueldo(){
    return sueldobase;
}
}

package sobreescribe;

public class Encargado extends Empleado {
public int getSueldo (){
   return sueldobase * 1,1;
}
}
```

- **2.** Compilará el siguiente código:

```
class prueba {
    protected String nombre;
    protected int ID;
    public String getIdent(){ return nombre; }
    public int getIdent(){ return ID; }
}
```

En caso de que no compile expón las razones.

Solución:

En el código anterior parece que se quiere hacer una especie de sobrecarga del método *getIdent()* pero la clase no compilará dado que el compilador se va a encontrar dos métodos con el mismo nombre y con la misma lista de parámetros, con lo cual, para él va a ser una implementación repetida. Concretamente con el compilador Geany dará el siguiente error:

```
prueba.java:5: getIdent() is already defined in prueba
```

Recuerda que para la sobrecarga de un método era necesario cambiar la lista de argumentos del mismo.

- **3.** Realiza una función que dada la fecha de nacimiento de una persona indique cuántos años tiene.

Solución:

```
public static int edad(String fecha_nac) {
   //Importante: fecha_nac tiene que tener el formato dd/MM/yyyy
   java.util.Date hoy = new Date(); //Fecha actual
   String[] tokens = fecha_nac.split("/");
     Calendar cal = new GregorianCalendar(Integer.parseInt(tokens[2]), Integer.
parseInt(tokens[1])-1, Integer.parseInt(tokens[0]));
   //Se resta 1 porque los meses comienzan en 0
   java.sql.Date fecha = new java.sql.Date(cal.getTimeInMillis());
   long diferencia = ( hoy.getTime() - fecha.getTime() )/(24 * 60 * 60 * 1000);
   //Se divide por los milisegundos que tiene un día. Se obtiene la diferencia en días

   return (int)diferencia/365;
}
```

- **4.** Tenemos una clase con un método *metodox()* que debe devolver un valor entero y da problemas al compilar. Parte del cuerpo de la clase es el siguiente:

```
    int dato;
    ......
    public int metodox(){
```

```
        return dato * 1.1;
    }
```

Se pide la reescritura del método utilizando un *wrapper Double* que solvente el problema de compilación.

Solución:

```
    public int metodox(){
        Double d = new Double(dato*1.1);
        return d.intValue();
    }
```

5. Realiza una clase huevo que esté compuesta por dos clases internas, una clara y otra yema. Crea dos métodos *hazYema()* y *hazClara()* que generen objetos de las clases yema y clara respectivamente. Realiza un método main en el que se creen objetos de cada una de las clases.

Solución:

```
public class huevo {
    class yema {
        yema() { System.out.println("Inicializando yema"); }
    }
    class clara {
        clara() { System.out.println("Inicializando clara"); }
    }
    huevo() { System.out.println("inicializando huevo"); }

    yema hazYema() {
        return new yema();
    }
    clara hazClara() {
        return new clara();
    }
    public static void main(String[] args) {
        huevo h = new huevo();
        yema y = h.hazYema();
        clara c = h.hazClara();
    }
}
```

- **6.** Averigua si son verdaderas o falsas las siguientes afirmaciones:
  - Una clase abstracta es una clase que no se puede instanciar.
  - Una clase abstracta es una clase que se usa únicamente para definir superclases.
  - Los métodos de las clases abstractas no tienen implementación.
  - En la declaración de una interfaz solamente pueden aparecer declaraciones de método y atributos pero nunca implementación de métodos.
  - Las interfaces no encapsulan datos.
  - Las interfaces pueden definir constantes simbólicas.

  La solución a este ejercicio se encuentra al final de los ejercicios propuestos.

- **7.** Implementa la siguiente estructura de clases.

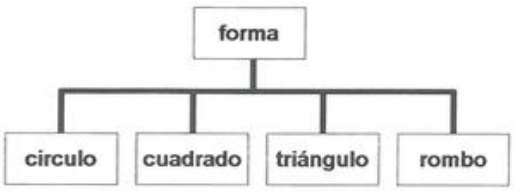


Figura 5.4. Jerarquía de clases desdencientes de la clase forma

Esta jerarquía deberá de tener las siguientes características:

- **La clase forma deberá de ser abstracta.**
- **La clase forma tendrá el método abstracto toString().**
- **La clase forma tendrá un método identidad que muestre el identificador interno de la clase.**
- **Círculo, cuadrado, triángulo y rombo descienden de la clase forma.**
- **Estas clases implementarán el método abstracto de la clase padre.**

Solución:

```
abstract class forma {
    void identidad() { System.out.println(this); }
    abstract public String toString();
}

class circulo extends forma {
    public String toString() { return "círculo"; }
}

class cuadrado extends forma {
```

```
    public String toString() { return " cuadrado "; }
}

class triangulo extends forma {
    public String toString() { return "triángulo"; }
}

class rombo extends forma {
    public String toString() { return "rombo"; }
}
```

8. En el siguiente ejercicio puedes ver cómo se utiliza el upcasting y downcasting, pero...

```
class testforma {
    public static void main(String[] args) {
            forma f = new circulo();
            f.identidad();
            circulo c = new circulo();
            ((forma)c).identidad();
            ((circulo)f).identidad();
            forma f2 = new forma();
            f2.identidad();
            (forma)f.identidad();
    }
}
```

- Para completar el ejercicio deberás de hacer lo siguiente:
  - Modificar la sintaxis de las líneas que dan problema.
  - Eliminar aquellas líneas que aunque sean sintácticamente correctas nunca pueden funcionar.

Solución:

```
class testforma {
    public static void main(String[] args) {
            forma f = new circulo();
            f.identidad();
            circulo c = new circulo();
            ((forma)c).identidad();
            ((circulo)f).identidad();
            //forma f2 = new forma();
            //f2.identidad();
//Las clases abstractas nunca pueden ser instanciadas
```

```
            ((forma)f).identidad();
        }
    }
```

¿Qué mostrará el programa por pantalla tras los cambios?

Solución:

círculo

círculo

círculo

círculo

EJERCICIOS PROPUESTOS

1. Averigua sin ejecutar el código qué mostrará el siguiente programa por pantalla:

```
public class bebe {
    static void pedir() {
            System.out.println(str1 + " , " + str2 + " , " + str3);
    }
    static {
            str2 = "mama pipi";
            str3 = "mama agua";
    }
    bebe() { System.out.println("Nacimiento del bebe"); }
    static String str2, str3,str1 = "papa tengo caca";
    public static void main(String[] args) {
            System.out.println("El bebe se ha despertado y va a pedir cosas");
            System.out.println("El bebe dice: " + bebe.str1);
            bebe.pedir();
    }
    static bebe bebe1 = new bebe();
    static bebe bebe2 = new bebe();
    static bebe bebe3 = new bebe();
}
```

Una vez que tengas claro lo que el programa debería de mostrar por pantalla ejecuta el código y verifica que lo que has pensado se cumple.

2. ¿Compilará el siguiente programa?

En caso afirmativo averigua sin ejecutar el código qué mostrará por pantalla:

```
public class bebe {
    static void pedir(String... argumentos) {
          for(String str : argumentos) System.out.println(str);
    }
    public static void main(String[] args) {
          pedir("mama pipi", "mama caca", "mama agua");
          pedir(new String[]{"papa jugar", "mama me aburro", "papa sed", "papa dormir",
"mama tengo hambre"});
    }
}
```

3. Tenemos la siguiente clase:

```
public abstract class sorteo{
    protected int posibilidades;
    public abstract int lanzar();
}
```

Se pide:

- Crear la clase *dado*, la cual desciende de la clase *sorteo*. La clase *lado*, en la llamada *lanzar()* mostrará un número aleatorio del 1 al 6.
- Crear la clase *moneda*, la cual desciende de la clase *sorteo*. Esta clase en la llamada al método *lanzar()* mostrará las palabras *cara* o *cruz*.

4. Realiza una clase *conversor* que tenga las siguientes características:

- Toma como parámetro en el constructor un valor entero.
- Tiene un método getNumero que dependiendo del parámetro devolverá el mismo número en el siguiente formato:

Parámetro	Formato
B	Binario (String)
H	Hexadecimal (String)
O	Octal (String)

- Realiza un método *main* en la clase para probar todo lo anterior.

- **5.** Realiza una clase *conversorfechas* que tenga los siguientes métodos:
  - *String* **normalToAmericano**(*String*). Este método convierte una fecha en formato normal dd/mm/yyyy a formato americano mm/dd/yyyy
  - *String* **americanoToNormal**(*String*). Este método realiza el paso contrario, convierte fechas en formato americano a formato normal.
- **6.** Estoy intentando hacer lo siguiente en mi programa:

```
final String s1=new String("Hola");
String s2=new String(" Mundo");
s1=s1+s2;
```

  - El compilador muestra un mensaje de error. ¿Qué podrá estar sucediendo?

- **7.** Tenemos la siguiente clase:

```
public abstract class vehiculo{
    private int peso;
    public final void setPeso(int p){peso=p;}
    public abstract int getVelocidadActual();
}
```

  - ¿Podrá tener descendencia esta clase?
  - ¿Se pueden sobrescribir todos sus métodos?
  - Razona tus respuestas.

- **8.** Realiza una clase *Test* en con un método *main* que tome por teclado dos números y muestre la suma, multiplicación, división y módulo. En el caso de que el segundo número sea 0, el programa deberá de atrapar las excepciones que se puedan producir. Para la resolución de este problema necesitarás utilizar *wrappers*.

- **9.** Diseña una clase con un método que permita averiguar la última cifra de un número introducido por teclado. Para la resolución de este problema deberás utilizar wrappers de tipos numéricos.

Solución ejercicio resuelto número 6:

- (Verdadera) Una clase abstracta es una clase que no se puede instanciar.
- (Falsa) Una clase abstracta es una clase que se usa únicamente para definir superclases.
- (Falsa) Los métodos de las clases abstractas no tienen implementación.
- (Falsa) En la declaración de un interfaz solamente pueden aparecer declaraciones de método y atributos pero nunca implementación de métodos.
- (Verdadera) Las interfaces no encapsulan datos.
- (Verdadera) Las interfaces pueden definir constantes simbólicas.

6 Lectura y escritura de información

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer todas las clases relativas a flujos.
- ✓ Valorar la importancia de la persistencia.
- ✓ Almacenar datos y objetos de forma definitiva utilizando ficheros.
- ✓ Recuperar datos y objetos de ficheros.
- ✓ Diseñar aplicaciones con interfaz gráfica.
- ✓ Construir y controlar los eventos producidos en aplicaciones con interfaz gráfica.

Cuando se realiza un programa como los que hemos hecho hasta ahora, los datos los proporciona el usuario o estarán en el programa y éste devolverá una serie de datos como respuesta a los datos recibidos. Generalmente, los programas leen datos y los almacenan en estructuras persistentes como bases de datos o ficheros. Las bases de datos son estructuras más complejas y avanzadas que los ficheros. Una vez almacenados esos datos los hacemos disponibles para otros programas o para el mismo programa, con lo cual se hacen estructuras muy necesarias en programación.

A FONDO

CONCEPTO DE REGISTRO

Cuando se almacenan datos en los ficheros aparece el concepto de registro. Un registro es una agrupación de datos. Por ejemplo, en un programa que gestione alumnos, un registro podría ser la agrupación de (número de matrícula, nombre, apellidos y curso). Cada uno de estos elementos se denomina campo. Estos campos pueden ser tipos elementales (int, char, etc.) o bien pueden ser a su vez otras estructuras de datos.

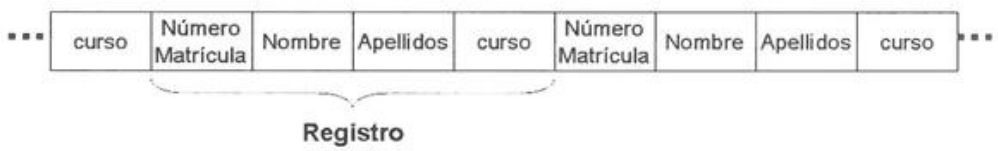


Figura 6.1. Estructura de un registro

Los registros, como se puede observar en la figura anterior, son una agrupación generalmente heterogénea de campos (por ejemplo el número de matrícula será diferente a nombre, el primero es numérico y el segundo será alfabético) tanto por sus tipos de datos como por su contenido. No obstante, pueden existir registros que contengan solamente un campo y, en ese caso, la estructura sería homogenea.

Como en este libro nos estamos centrando en Programación Orientada a Objetos, lo normal es que en un fichero almacenemos objetos y no registros, y que en vez de almacenar campos almacenemos atributos.

Una vez visto en este capítulo los ficheros y flujos de datos como concepto previo a los ficheros, veremos las *interfaces* de usuario en Java y la generación de programas en entorno gráfico.

6.1 FLUJOS DE DATOS

Los flujos de datos son flujos de información entre el programa y el origen o destino de la información. Los flujos de información se tratan en Java mediante objetos *stream* (flujo en inglés), los cuales harán de intermediario entre el programa y ese origen o destino de la información. Los flujos servirán entonces para abstraer y simplificar la programación. Los programas se encargan de leer y escribir en los flujos sin importarles dónde se leen y se escriben los datos.

Los programas, cuando quieren leer datos de un fichero, lo primero que hacen es abrir un flujo de entrada y leen la información que contiene el fichero mediante el flujo de datos de entrada. Para grabar datos la operación es similar, se abre un flujo de salida y el programa va escribiendo los datos en el flujo de salida y de esta forma se almacenan los datos.

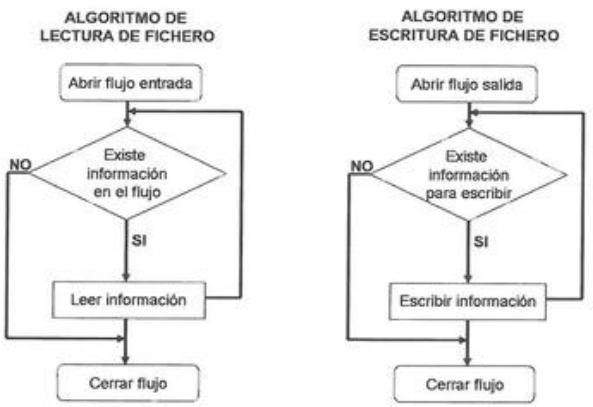


Figura 6.2. Algoritmos de lectura y escritura de ficheros

Generalmente, la forma de leer y escribir datos en un fichero sigue unos algoritmos similares: se abre el flujo, se realizan una serie de operaciones y por último se cierra el flujo. En la figura anterior se muestran los algoritmos de lectura y escritura de datos en un fichero que como se puede ver son muy similares.

6.2 CLASES RELATIVAS A FLUJOS

El paquete java.io está en la biblioteca estándar de Java y es el que tiene todas las clases necesarias para leer y escribir datos en flujos y en el sistema de ficheros. Este paquete tiene una serie de interfaces, clases y excepciones, todas relacionadas con la entrada/salida de datos.

Importante
Si se le pasa un *null* como argumento a un constructor o método en las clases o interfaces de este paquete, el programa lanzará una excepción del tipo *NullPointerException*.

En la siguiente figura se muestra la jerarquía de clases del paquete java.io. Como se puede apreciar, estas clases se dividen en dos grupos. El grupo de la derecha (las clases abstractas InputStream y OutputStream y sus subclases) está pensado para trabajar con datos de tipo byte, mientras que el grupo de la izquierda (las clases abstractas *Reader* y *Writer* y sus subclases) está pensado para trabajar con datos de tipo *char*. Las interfaces de ambos grupos son muy parecidas.

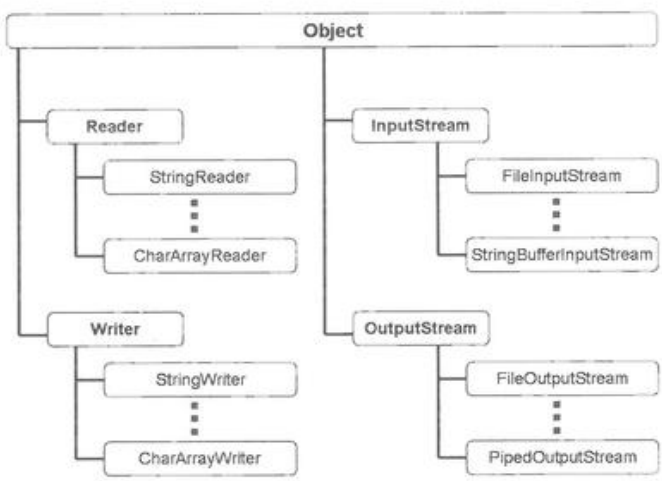


Figura 6.3. Jerarquía de las clases Reader, Writer, InputStream y OutputStream

Las tuberías
Las tuberías o *pipes* son flujos de datos que permiten conectar dos programas o procesos entre sí transmitiéndose información entre uno y otro. Su función es canalizar la salida de un programa para que sirva como entrada de otro programa.

La siguiente tabla muestra las subclases que permiten leer y escribir caracteres o bytes en un vector o matriz en memoria, en un archivo o en una tubería o *pipe*:

Tabla 6.1. Medios y flujos asociados a los mismos

Medio	Flujo de caracteres	Flujo de bytes
Memoria	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
	StringReader StringWriter	StringBufferInputStream
Archivo	FileReader FileWriter	FileInputStream FileOutputStream
Pipes/Tuberías	PipedReader PipedWriter	PipedInputStream PipedOutputStream

El siguiente ejemplo muestra la utilización de la subclase StringReader y la clase CharArrayWriter:

```
import java.io.*;
public class EjemploFlujos
{
  public static void main(String[] args)
  {
    String s = new String("En un lugar de la mancha de cuyo nombre no quiero acordarme,
");
    s = s + "no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, ";
    s = s + "adarga antigua, rocín flaco y galgo corredor...";
    char[] arr = new char[s.length()];
    int car = 0;
    StringReader flujoInput = new StringReader(s);
     flujoOutput = new CharArrayWriter();
    try
    {
         while ((car = flujoInput.read()) != -1){
           flujoOutput.write(car);
         }
         arr = flujoOutput.toCharArray();
         System.out.println(arr);
    }
    catch (IOException e) {
         e.printStackTrace();
    }
    finally
    {
```

```
            flujoInput.close();
            flujoOutput.close();
        }
    }
}
```

El programa anterior crea un *String* con los datos que se quieren procesar y un array con la misma longitud que el *String* anterior. Se utilizará un objeto flujoInput de la clase *StringReader* para leer el *String* carácter a carácter. Los caracteres leidos se van almacenando en el objeto *FlujoOutput* uno a uno hasta que se lee el carácter -1 que indica el final del *String*. Una vez se han leido todos los caracteres se copian al array *arr* mediante el método *toCharArray()* del objeto *flujoOutput* de la clase *CharArrayWriter*. Por último y no menos importante se cierran los flujos abierto mediante los método *close()* de cada uno de ellos.

6.3 UTILIZACIÓN DE FLUJOS

Existen clases que **alteran** el comportamiento de un *stream* ya definido, estas clases pueden añadir un *buffer*, realizar una conversión, añade un filtro, etc. Estas clases son las siguientes:

Tabla 6.2. Clases de flujos

Clases	Características o función que pueden utilizar
BufferedReader(C) BufferedWriter(C) BufferedInputStream(B) BufferedOutputStream(B)	Tienen la propiedad de añadir un *buffer* al funcionamiento del objeto. En el siguiente ejemplo se puede observar la utilización del método *readLine()* el cual hace más eficiente el uso de estos objetos.
InputStreamReader(C) OutputStreamWriter(C)	Utilizadas como clases puentes las cuales transforman streams que utilizan bytes en otros que utilizan caracteres.
ObjectInputStream(B) ObjectOutputStream(B)	Tienen la propiedad de la seriación o serialización.
FilterReader(C) FilterWriter(C) FilterInputStream(B) FilterOutputStream(B)	Pueden aplicar filtros a los *stream* de datos.

DataInputStream(B) DataOutputStream(B)	Tienen la capacidad de la transformación de datos. Leen y escriben en formatos propios de Java y facilitan las transmisiones entre equipos de distinto funcionamiento.
PushbackReader(C) PushbackInputStream(B)	Tienen la capacidad de poder mirar cuál es el siguiente carácter de la entrada y devolverlo.
PrintWriter(C) PrintStream(B)	Tienen métodos para imprimir las variables Java con apariencia normal.
SequenceInputStream(B)	Tienen la capacidad de la concatenación.

(C) Operan con flujos de caracteres

(B) Operan con flujos de bytes

A continuación vamos a ver ejemplos simples de algunas de las clases anteriores.

En el siguiente programa se muestra como se utiliza las clases **InputStreamReader** y **BufferedReader**:

```
import java.io.*;
public class EjemploLecturaPorConsola {
public static String leercadena(){
  String cad="";
  BufferedReader br;
  br = new BufferedReader(new InputStreamReader(System.in));
  try{
    cad = br.readLine();
  }catch (IOException e ) {
      e.printStackTrace();
  }
  return cad;
}
 public static void main(String args[]) throws IOException {
  String cad;
  System.out.println("Este programa hace eco hasta que escribas para");
  do {
   cad = leercadena();
   System.out.println(cad);
  } while(!cad.equals("para"));
 }
}
```

El ejemplo anterior lee cadenas por teclado y las repite por pantalla hasta que el usuario teclea "para".

La línea más interesante del programa anterior es la siguiente:

```
br = new BufferedReader(new InputStreamReader(System.in));
```

En esta línea se crea un objeto *BufferedReader* que contiene el método *readLine()*, el cual devuelve una línea completa del *buffer*. Este objeto en su constructor pide un *Reader*, para lo cual utilizamos un objeto *InputStreamReader* que a su vez, pide en su constructor un InputStream(el objeto System.in).

Otro ejemplo de utilización de flujos es el siguiente. En este caso se utiliza el objeto PrintWriter:

```
import java.io.*;
public class EscrituraPorPantalla {
  public static void main(String args[]) throws IOException {
    PrintWriter pantalla = new PrintWriter(System.out);
    char[] array = { 'M', 'o', 'r', 'e', 'n', 'o' };
    String str = new String("Juan Carlos");
    pantalla.write(str);
    pantalla.print(" ");
    pantalla.write(array, 0, 6);
    pantalla.println("");
    pantalla.flush();
  }
}
```

En el programa anterior se utiliza el objeto **PrintWriter** el cual es redirigido a la pantalla. También se utilizan los métodos *write*, *print*, *println* y *flush* de este método. El método *flush* vacia el *buffer* de contenido.

A FONDO

LA CLASE PUSHBACKREADER

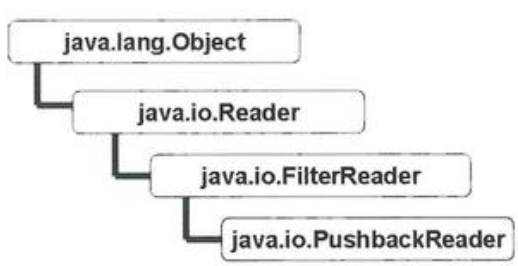


Figura 6.4. Estructura jerárquica de la clase PushbackReader

Esta clase permite devolver un carácter al flujo de entrada. Imaginemos que estamos leyendo caracteres por teclado, esta clase sería capaz de leer los caracteres de teclado y devolverlos a su *buffer* si hubiese la necesidad. Para devolver los caracteres al flujo utiliza el método *unread()*. Este método

es polimórfico y tiene varias formas de invocación:

```
void unread(int ch)
void unread(char buffer[ ])
void unread(char buffer[ ], int offset, int numChars)
```

En el primero de los casos solamente se puede devolver un carácter en cada llamada al método. En el segundo de los casos se puede devolver un array de caracteres (variable *buffer*). El tercero de los métodos es más sofisticado pues permite devolver una serie de caracteres (numChars) de un array (*buffer*) comenzando en una posición determinada (*offset*).

Hay que tener en cuenta que el buffer que recoge los datos devueltos tiene un límite, si se sobrepasa ese límite, el programa lanzará una excepción del tipo IOException.

La clase PushbackReader puede ser creada mediante dos constructores distintos:

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

En el segundo de los casos, además de encargarse el objeto de crear el *buffer* para devolver los datos, se le especifica el tamaño que tiene que tener éste.

En el siguiente ejemplo se muestra un caso práctico de la utilización de un objeto *PushbackReader*. En el siguiente programa se sustituyen los operadores decremento (b++) por una asignación y una suma (b=b+1), teniendo en cuenta que las variables están definidas solo con una letra. Nótese que cuando el programa se encuentra con un carácter '+' mira en el flujo si hay otro carácter '+' y, si no es así, lo devuelve al flujo mediante el método *unread()*.

```
import java.io.*;
class ejemploPushbackReader {
public static void main(String args[]) throws IOException {
    String sentencias = "a=a+1;c++;b+=5;c=a+b;b++;";
    StringReader sr = new StringReader(sentencias);
    PushbackReader pbr = new PushbackReader(sr);
    int ultimo=pbr.read(),penultimo=0;
    while (ultimo != -1) {
    switch(ultimo) {
    case '+':
    if ((ultimo = pbr.read()) == '+'){
    System.out.print("="+(char) penultimo+"+1");
    }else{
    pbr.unread(ultimo);
    System.out.print('+');
    }
    break;
    case ';':
    System.out.println((char)ultimo);
    break;
    default:
    System.out.print((char) ultimo);
```

```
        }
        penultimo=ultimo;
        ultimo = pbr.read();
        }
    }
}
```

6.4 FICHEROS DE DATOS

Cuando queremos que los datos con los que estamos trabajando perduren en el tiempo necesitamos almacenarlos en 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 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.

6.4.1 LECTURA Y ESCRITURA SECUENCIAL EN UN ARCHIVO

La escritura y lectura secuencial de un fichero se puede realizar en Java utilizando las clases **FileInputStream** y **FileOutputStream**.

Escribir datos en un archivo

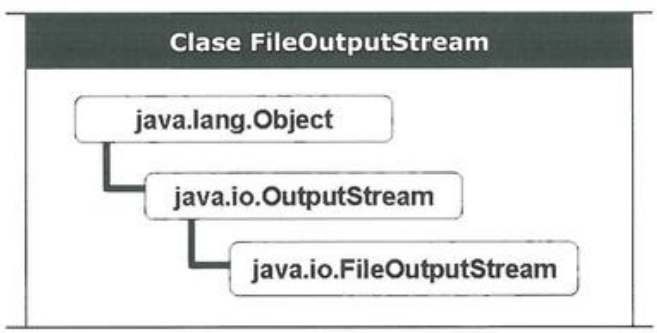


Figura 6.5. Estructura jerárquica de la clase FileOutputStream

Para escribir datos en un archivo se puede utilizar la clase *FileOutputStream* la cual hereda de *OutputStream* y permite escribir bytes en un fichero. En el siguiente ejemplo se puede ver una utilización de la misma:

```
import java.io.*;
public class TestFichero{
    public static void main(String[] args){
          FileOutputStream f=null;
          String s = "En un lugar de la mancha de cuyo nombre no quiero acordarme...";
          char c=0;
          try{
                f=new FileOutputStream("datos.txt");
                for (int i=0; i<s.length();i++){
                      c=s.charAt(i);
                      f.write((byte)c);
                }
          }catch(IOException e){
                e.printStackTrace();
          }finally{
                try{
                      f.close();
                }catch(IOException e){
                      e.printStackTrace();
                }
          }
    }
}
```

Como se puede observar, en el código anterior se crea un objeto de la clase *FileOutputStream* llamado *f* y se inicializa a *null*. En el siguiente código se crea una instancia del objeto llamando al constructor:

```
f=new FileOutputStream("datos.txt");
```

Como se puede observar, se crea una instancia del objeto y se le indica el nombre del fichero que va a crear. Si no se le indica ninguna ruta, el fichero se creará en el mismo directorio donde reside el programa. Existen tres constructores para la clase *FileOutputStream*:

- FileOutputStream(String nombre_fichero);
- FileOutputStream(String nombre_fichero, boolean añadir);
- FileOutputStream(File fichero);

En el segundo constructor el *FileOutputStream* permite añadir datos al fichero si se invoca con el parámetro añadir = *true*. El tercer constructor utiliza un objeto de la clase *File*, el cual se estudiará más adelante.

Obsérvese que nuestra información está almacenada en un objeto de tipo *String* y la clase *FileOutputStream* solamente escribe bytes al fichero. Para eso se recorre el *String* y se van extrayendo los caracteres invocando al método s.charAt(i). De esa forma sí es posible insertar los datos al fichero.

Como buena práctica de programación se recomienda cerrar los flujos una vez que han sido utilizados. Para cerrar el flujo bastaría con ejecutar la sentencia *f.close()*, pero vemos que hemos elegido otro código:

```
try{
        f.close();
}catch(IOException e){
        e.printStackTrace();
}
```

Esto es porque la invocación al método *close()* puede lanzar una IOException y nuestro código no debería de dejar de atraparla (obsérvese que se encuentra en el bloque *finally*). No obstante, para mejorar el código anterior sería mejor modificar la sentencia *f.close()* por la siguiente:

```
if (f != null) f.close();
```

De esa manera solo se ejecutará el método *close* cuando *f* sea distinto de *null*, lo cual es más apropiado.

Leer datos de un archivo

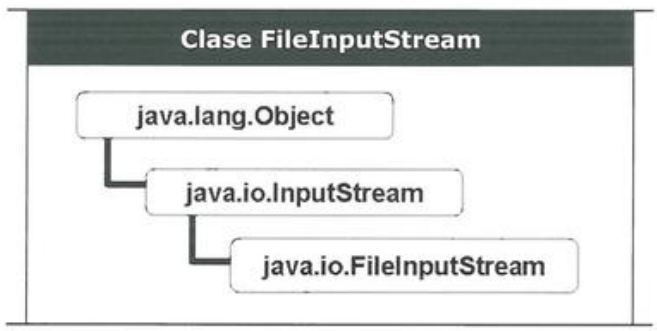


Figura 6.6. Estructura jerárquica de la clase FileInputStream

La lectura de los datos de un fichero es un proceso parecido al anterior. En el siguiente ejemplo el programa lee el fichero de datos escrito en el programa anterior y lo muestra por pantalla. Para leer bytes de un fichero utilizamos la clase *FileInputStream* que hereda de la clase *InputStream*. Sus constructores son los siguientes:

- FileInputStream(String nombre_fichero);
- FileInputStream(File fichero);

```
import java.io.*;
public class TestFichero{
    public static void main(String[] args){
            FileInputStream f=null;
            String s="";
            char c;
```

```
            try{
                  f=new FileInputStream("datos.txt");
                  int size = f.available();
                  for (int i=0;i<size;i++){
                        c=(char)f.read();
                        s=s+c;
                  }
            }catch(IOException e){
                  e.printStackTrace();
            }finally{
                  System.out.println(s);
                  try{
                        f.close();
                  }catch(IOException e){
                        e.printStackTrace();
                  }
            }
      }
}
```

6.4.2 LA CLASE FILE

Recuerda
El objeto *File* puede trabajar tanto con ficheros como con directorios.

La clase *File* desciende directamente de la clase *Object*. El arbol de herencia de la clase *File* es el siguiente

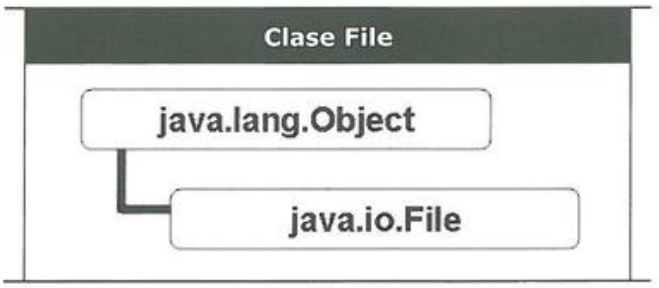


Figura 6.7. *Estructura jerárquica de la clase File*

El uso de la clase *File* es la manera más útil de trabajar con ficheros. Un objeto de la clase *File* representa un fichero (o un directorio) del sistema de archivos. La clase *File* tiene múltiples métodos que nos van a permitir realizar todo tipo de operaciones con los ficheros. Comencemos a comentar cada uno de ellos:

Tabla 6.3. Constructores de la clase File

Constructores	Descripción
public File (String ruta_absoluta)	Este constructor crea un objeto *File* al cual hay que pasarle toda la ruta incluido el nombre de archivo.
public File(String ruta, String nombre)	En este constructor hay que pasarle la ruta relativa y el nombre del fichero.
public File(File ruta, String nombre)	Como el objeto *File* puede representar un directorio o un fichero, el primer parámetro funcionará como directorio (ruta relativa) y el segundo parámetro será el nombre del fichero a crear.

Ejemplos de creación de objetos de la clase *File*:

```
//primer constructor
File f1 = new File("/home/jcmoreno/datos.txt");
//segundo constructor
File f2 = new File("/home/jcmoreno","datos.txt");
//tercer constructor
File dir = new File("/home/jcmoreno");
File f3 = new File(dir,"datos.txt");
```


Importante
En Windows®, el separador de directorios "\" hay que duplicarlo "\\" ya que se especifica mediante una secuencia de escape.

A continuación se muestran algunos de los métodos del objeto *File*, existen muchos más:

Tabla 6.4. Métodos de la clase File

Métodos	Descripción
public boolean isDirectory()	Estos dos métodos sirven para saber si se está trabajando con ficheros o con directorios.
public boolean isFile()	
public boolean exists()	Con este método nos cercioramos si el fichero existe realmente.
public boolean delete()	Permite borrar el fichero o directorio al que referencia el objeto *File*.

public boolean renameTo(Filedest)	Permite renombrar el fichero o directorio al que referencia el objeto *File*. El nuevo nombre lo toma como parámetro.
public boolean canRead()	Estos métodos nos permiten averiguar si tenemos permisos de lectura y escritura sobre el fichero.
public boolean canWrite()	
public String getPath()	Devuelve la ruta relativa del archivo.
public String getAbsolutePath()	Devuelve la ruta absoluta del archivo (incluye el nombre del archivo).
public String getName()	Devuelve el nombre del archivo.
public String getParent()	Devuelve el directorio padre (del que cuelga el fichero o directorio).
public long length()	Devuelve el tamaño del archivo en bytes, si lo que se quiere son los kilobytes habrá que dividirlo entre 1024 y si se quieren los megabytes dividirlo nuevamente por 1024.
public boolean equals(Objectobj)	Este método permitirá comparar objetos *File* para saber si son iguales o no.

Los siguientes métodos son utilizados cuando se trabaja con directorios:

Tabla 6.5. Métodos utilizados en el trabajo con directorios

Métodos para directorios	Descripción
public boolean mkdir()	Crea un directorio.
public String[] list()	Devuelve un array de objetos *String* con los nombres de los ficheros/directorios del directorio al que apunta el objeto *File*.
public String[] list(filtro)	Devuelve un array de objetos *String* con los nombres de los ficheros que cumplen con un determinado filtro (por ejemplo "*.java").
public File[] listFiles()	Devuelve un array de objetos *File* con los ficheros/directorios del directorio al que apunta el objeto *File*.

El siguiente ejemplo muestra la utilización de algunos de los métodos de la clase *File*. En este ejemplo se analizará si el directorio en cuestión existe, mostrará si se tiene permiso de lectura y escritura y se listarán los ficheros y directorios que contiene:

```
import java.io.File;
public class TestFichero{
    public static void main(String[] args){
          File dir = new File("/home/jcmoreno/programacion");

          if (dir.exists()) {
```

```
                System.out.println("Existe el directorio "+dir.getName());
            }else{
                System.out.println("El directorio no existe");
            }
            if (dir.canRead())
               System.out.println("El directorio existe y tiene permiso de lectura");
            if (dir.canWrite())
               System.out.println("El directorio existe y tiene permiso de escritura");
            File[] ficheros = dir.listFiles();
            for (File f : ficheros)
                System.out.println(f.getName());
    }
}
```

El caso anterior la sentencia:

```
File dir = new File("/home/jcmoreno/programacion");
```

Demuestra que se está trabajando con un sistema de ficheros tipo Unix. En sistemas Windows®, esta sentencia podría ser parecida a la siguiente:

```
File dir = new File("C:\\Documents and Settings\\JUAN CARLOS\\Escritorio\\
programacion");
```

Todo depende del directorio con el que queramos trabajar.

6.4.3 CLASES FILEWRITER Y FILEREADER

Ambas clases, *FileWriter* y *FileReader*, trabajan con flujos de caracteres (*char*) en modo lectura y escritura sobre un fichero. Los árboles de herencia para ambas clases son los siguientes:

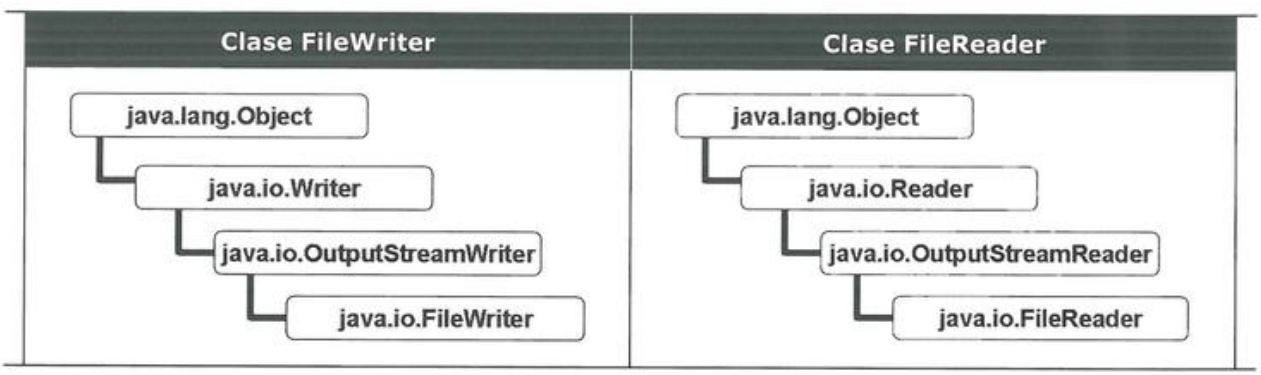


Figura 6.8. Estructura jerárquica de la clase FileWriter

Figura 6.9. Estructura jerárquica de la clase FileReader

Los constructores para la clase **FileWriter** son:

Tabla 6.6. Constructores de la clase FileWriter

Constructores	Descripción
FileWriter(File file)	Dado un objeto *File* construye un objeto *FileWriter*.
FileWriter(File file, boolean append)	Dado un objeto *File* construye un objeto *FileWriter* para añadir datos.
FileWriter(FileDescriptor fd)	Crea un objeto *FileWriter* asociado a un descriptor de fichero. Un *FileDescriptor* es la representación a más bajo nivel de un fichero, dispositivo o *socket*.
FileWriter(String fileName)	Construye un objeto *FileWriter* a partir del nombre de un fichero.
FileWriter(String fileName, boolean append)	Construye un objeto *FileWriter* para añadir datos a partir del nombre de un fichero.

Los constructores para la clase **FileReader** son:

Tabla 6.7. Constructores de la clase FileReader

Constructores	Descripción
FileReader(File file)	Crea un objeto *FileReader* dado un fichero desde el que leer.
FileReader(FileDescriptor fd)	Crea un objeto *FileReader* dado un FileDescriptor o descriptor de fichero desde el que leer.
FileReader(String fileName)	Crea un objeto *FileReader* dado el nombre de un fichero.

El siguiente código muestra un ejemplo de la utilización de las clases *FileReader* y *FileWriter*:

```
import java.io.*;
public class TestFichero{
    public static void main(String[] args){
          String  []amigos={"Andrés  Rosique","Pedro  Ruiz","Isaac  Sanchez","Juan
Serrano"};

          File fs = new File("amigos.txt");
          try{
                FileWriter fw = new FileWriter(fs);
```

```
                for (String s : amigos){
                        fw.write(s,0,s.length());
                        fw.write("\r\n");
                }
                if (fw != null) fw.close();
        }catch(IOException e){
                e.printStackTrace();
        }
        File fe = new File("amigos.txt");
        if (fe.exists()){
                try{
                        FileReader fr = new FileReader(fe);
                        BufferedReader br = new BufferedReader(fr);
                        String s;
                        while((s = br.readLine()) != null) {
                                System.out.println(s);
                        }
                        if (fr != null) fr.close();
                }catch(IOException e){
                        e.printStackTrace();
                }
        }
    }
}
```

El programa tiene almacenado una serie de personas en una estructura amigos que es un vector de objetos *String*. Los datos en este vector se graban en un fichero "amigos.txt", cada persona en una línea y para ello se graba en el fichero el retorno de carro y línea "\r\n" (\r es el retorno de carro y \n es el carácter de nueva línea).

Para la lectura del fichero se utiliza un objeto *FileReader*. Como queremos leer línea a línea utilizamos un objeto de la clase *BufferedReader* que tiene el método *readLine()*, que permite leer una línea de un flujo de entrada. La clase *BufferedReader* es un *reader*. Abajo se muestra sus ascendientes jerárquicos:

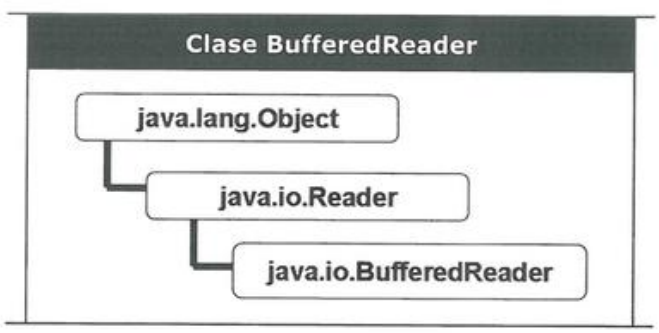


Figura 6.10. *Estructura jerárquica de la clase BufferedReader*

6.4.4 FLUJOS DE DATOS DATAOUTPUTSTREAM Y DATAINPUTSTREAM

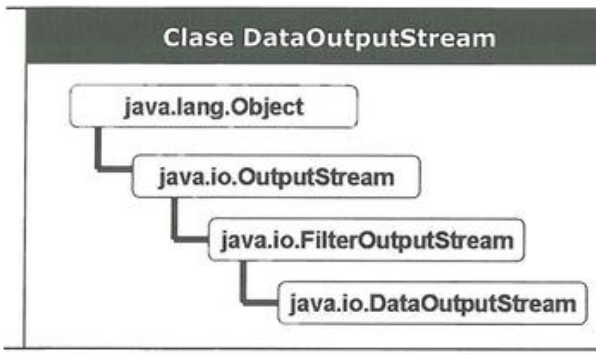


Figura 6.11. Estructura jerárquica de la clase DataOutputStream

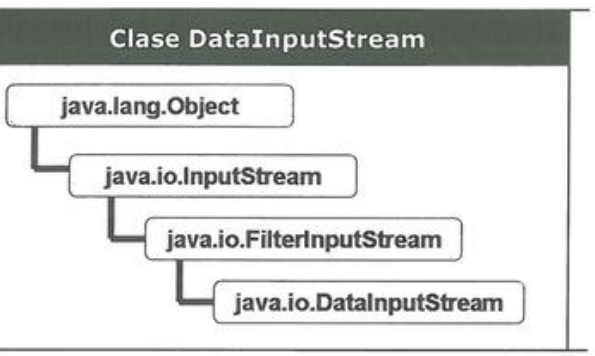


Figura 6.12. Estructura jerárquica de la clase DataInputStream

Hasta ahora hemos trabajado con ficheros de texto y realizado ejercicios de lectura y escritura de ficheros. ¿Qué más queremos? Pues bien, imaginemos que queremos almacenar primitivos en nuestro fichero (*boolean*, *float*, *long*, *int*, *double*, *byte* o *short*) y luego recuperarlos como tal. Con los mecanismos utilizados hasta ahora sería difícil realizar esto teniendo que hacer esta tarea de una manera poco natural y eficiente. Java en su paquete *Java.io* ofrece dos clases, *DataOutputStream* y *DataInputStream*, las cuales van a permitir escribir y leer los datos a los ficheros en formato UTF-8.

Formato UTF-8
El formato UTF-8 es un formato de 8 bits y codifica caracteres Unicode e ISO 10646 utilizando símbolos de longitud variable (1-4 bytes). Por sus características es un formato muy útil en la codificación de correos electrónicos y páginas web.

Como se puede observar, ambas clases descienden de una clase filtro (FilterInputStream y FilterOutputStream), lo cual es lógico, pues tienen que hacer la conversión/desconversión de los datos al formato UTF-8. El objetivo principal de las clases filtro en Java es la modificación/transformación de los datos.

La forma de trabajar con estas clases es la siguiente:

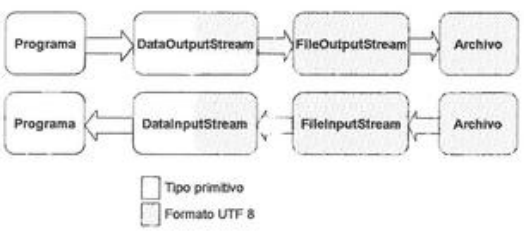


Figura 6.13. Trabajando con Streams

Como puede observarse, cuando se desea almacenar datos en un fichero se crea un objeto filtro *DataOutputStream*. En este objeto podemos insertar datos primitivos de todo tipo, pero si lo que queremos es insertarlos en un archivo, deberemos conectar dicho objeto a otro del tipo *FileOutputStream*. La lectura es un proceso análogo, el objeto *FileInputStream* nos va a servir para leer datos del archivo y pasárselos al objeto *DataInputStream* del cual vamos a poder recuperar la información.

En el siguiente programa se puede ver un ejemplo de utilización de estas clases:

```
import java.io.*;
public class TestFichero2{
    public static void main(String[] args){
          String  []amigos={"Andrés  Rosique","Pedro  Ruiz","Isaac  Sanchez","Juan
Serrano"};
          long []telefonos={653364787,627463746,644567346,623746348};
          //escritura del fichero
          try{
                FileOutputStream fs = new FileOutputStream("amigos.txt");
                DataOutputStream d = new DataOutputStream(fs);
                for (int i=0;i<4;i++){
                      d.writeUTF(amigos[i]);
                      d.writeLong(telefonos[i]);
                }
                if (d != null){
                      d.close();
                      fs.close();
                }
          }catch(IOException e){
                      e.printStackTrace();
          }
          //lectura del fichero
          try{
                File f=null;
                FileInputStream fe = null;
                DataInputStream d = null;
                try{
                      f = new File("amigos.txt");
                      if (f.exists()){
                            fe = new FileInputStream(f);
                            d = new DataInputStream(fe);
                            String s;
                            Long l;
                            while(true){
```

```
                                    s = d.readUTF();
                                    System.out.print(s+" -> ");
                                    l = d.readLong();
                                    System.out.println(l);
                              }
                        }
                  }catch (EOFException eof) {
                        System.out.println(" ---------------------------");
                  }catch (FileNotFoundException fnf) {
                        System.err.println("Fichero no encontrado " + fnf);
                  }catch(IOException e){
                        System.err.println("Se ha producido una IOException");
                        e.printStackTrace();
                  }catch (Throwable e) {
                        System.err.println("Error de programa: " + e);
                        e.printStackTrace();
                  }finally{
                        if (d != null) {
                              d.close();
                              fe.close();
                        }
                  }
            }catch(IOException e){
                  e.printStackTrace();
            }
      }
}
```

A continuación, se van a comentar las instrucciones más interesantes vistas en el programa anterior:

```
FileOutputStream fs = new FileOutputStream("amigos.txt");
DataOutputStream d = new DataOutputStream(fs);
```

En las sentencias anteriores, como se ha dicho antes, se crea un *FileOutputStream* para escribir datos a un fichero (asociado al fichero amigos.txt): También se crea un *DataOutputStream*, que está conectado al *FileOutputStream*, con lo cual cualquier dato que se escriba al *DataOutputStream* se escribirá directamente al fichero.

```
d.writeUTF(amigos[i]);
d.writeLong(telefonos[i]);
```

En estas dos sentencias escriben una cadena de caracteres y un dato de tipo *long*. La cadena de caracteres se almacenará en formato UTF-8. En este formato, los dos primeros bytes contienen el número de bytes de datos que ocupará la cadena de caracteres.

En la siguiente tabla se muestran algunos de los métodos de la clase DataOutputStream:

Tabla 6.8. Métodos de la clase DataOutputStream

Método	Descripción
writeBoolean	Escribe un dato de tipo boolean
writeByte	Escribe un dato de tipo byte
writeChar	Escribe un dato de tipo char
writeShort	Escribe un dato de tipo short
writeInt	Escribe un dato de tipo int
writeLong	Escribe un dato de tipo long
writeFloat	Escribe un dato de tipo float
writeDouble	Escribe un dato de tipo double
writeBytes	Escribe una cadena como una sucesión de bytes
writeChars	Escribe una cadena como una sucesión de caracteres
writeUTF	Escribe una cadena en el formato UTF-8

Como norma de buena programación se recomienda cerrar los flujos una vez utilizados. En las siguientes sentencias se muestra como:

```
if (d != null){
d.close();
    fs.close();
}
```

El proceso de lectura se realiza de manera análoga al proceso de escritura.

```
File f=null;
FileInputStream fe = null;
DataInputStream d = null;
...
f = new File("amigos.txt");
...
fe = new FileInputStream(f);
d = new DataInputStream(fe);
```

El siguiente paso a realizar una vez asociados el *FileInputStream* con el *DataInputStream* es la lectura de los datos del fichero hasta encontrar la marca de fin de fichero. Las siguientes sentencias explican cómo se realiza este proceso.

El alumno debe de observar que durante el proceso de lectura se ejecuta un bucle infinito (while(true)) hasta que el programa lance una excepción del tipo *EOFException*. Esta excepción será atrapada en el bloque *catch*.

```
while(true){
    s = d.readUTF();
    System.out.print(s+" -> ");
    l = d.readLong();
    System.out.println(l);
}
```

En la siguiente tabla se muestran algunos de los métodos de la clase *DataInputStream*:

Tabla 6.9. Métodos de la clase DataInputStream

Método	Descripción
readBoolean	Retorna un dato de tipo *boolean*
readByte	Retorna un dato de tipo *byte*
readChar	Retorna un dato de tipo *char*
readShort	Retorna un dato de tipo *short*
readInt	Retorna un dato de tipo *int*
readLong	Retorna un dato de tipo *long*
readFloat	Retorna un dato de tipo *float*
readDouble	Retorna un dato de tipo *double*
readUTF	Retorna una cadena de caracteres la cual está en UTF-8

Las siguientes sentencias muestran el bloque *catch* del tratamiento de excepciones. Como se puede observar se captura el fin de fichero en primer lugar como una excepción. En ese caso la lectura se realizó de forma correcta. Posteriormente se tratan las demás excepciones posibles.

```
catch (EOFException eof) {
    System.out.println(" ---------------------------");
}catch (FileNotFoundException fnf) {
    System.err.println("Fichero no encontrado " + fnf);
}catch(IOException e){
    System.err.println("Se ha producido una IOException");
    e.printStackTrace();
}catch (Throwable e) {
    System.err.println("Error de programa: " + e);
    e.printStackTrace();
}
```

6.5 ALMACENAMIENTO DE OBJETOS EN FICHEROS. PERSISTENCIA. SERIALIZACIÓN

La serialización de objetos (o seriación) consiste es transformar un objeto en una secuencia o serie de bytes de tal manera que se represente el **estado** de dicho objeto. Una vez que tenemos serializado el objeto se puede enviar a un fichero, se podría enviar por la red, etc. Si por ejemplo tenemos el objeto seriado y almacenado en un fichero sería posible recomponer el objeto.

El estado de un objeto es básicamente el estado de cada uno de los campos. Imaginemos que un campo es a su vez otro objeto, en ese caso debería de ser serializado para serializar el primer objeto.

Recuerda

Es posible no serializar algunos de los atributos de un objeto, esto se realiza utilizando el modificador *transient*. Por ejemplo:

```
protected transient int dato;
```

En este caso, el campo dato no interesa que sea persistente.

Para poder serializar un objeto de una clase es necesario que implemente la interfaz java.io.Serializable. En principio, la interfaz Serializable no define ningún método nuevo, luego el propósito de esto es marcar las clases que vamos a convertir en secuencias de bytes. Un ejemplo de esto es el siguiente:

```
public class Amigo implements java.io.Serializable{
// atributos y métodos de la clase
}
```

El objeto *Amigo* anterior se ha marcado ya como serializable, ahora Java se encargará de realizar la serialización de forma automática.

Para el almacenamiento y la recuperación de objetos utilizaremos las clases *ObjetctOutputStream* y *ObjectInputStream* cuya forma de utilización es análoga a las clases anteriormente vistas, *DataOutputStream* y *DataInputStream*.

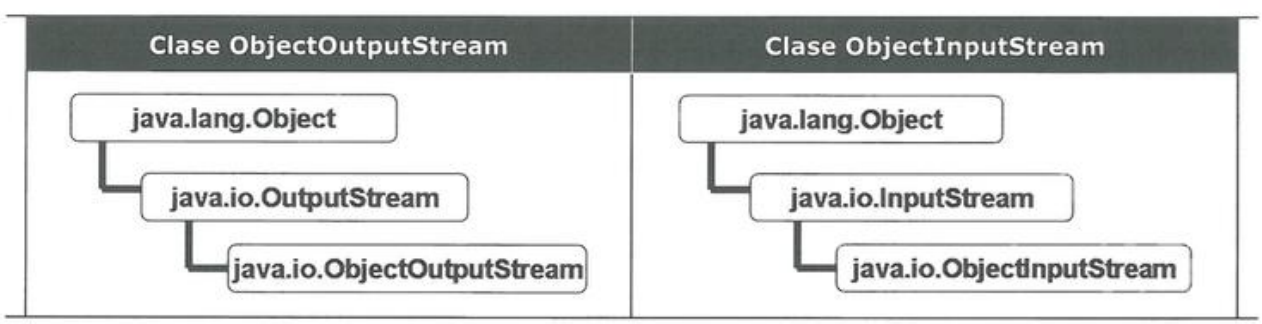


Figura 6.14. Jerarquía de la clase ObjectOutputStream

Figura 6.15. Jerarquía de la clase ObjectInputStream

A continuación, se muestra el código de un ejemplo de escritura y lectura de objetos en un fichero. Este ejemplo es muy parecido al visto en apartados anteriores nada más que en vez de datos se graban y leen objetos en un fichero (los cuales no dejan de ser datos).

```
public class amigo implements java.io.Serializable{
    protected String nombre;
    protected long telefono;
    public amigo(String n, long t){
                nombre = n;
                telefono = t;
    }
    public void print(){
          System.out.println(nombre + " -> " + telefono);
    }
}

import java.io.*;
public class TestFichero3{
public static void main(String[] args){
    String []amigos={"Andrés Rosique","Pedro Ruiz","Isaac Sanchez","Juan Serrano"};
    long []telefonos={653364787,627463746,644567346,623746348};
    //escritura del fichero
    try{
          FileOutputStream fs = new FileOutputStream("amigos.txt");
          ObjectOutputStream oos = new ObjectOutputStream(fs);
          for (int i=0;i<4;i++){
                amigo a = new amigo(amigos[i],telefonos[i]);
                oos.writeObject(a);
          }
          if (oos != null){
                oos.close();
                fs.close();
          }
    }catch(IOException e){
          e.printStackTrace();
    }
    //lectura del fichero
    try{
          File f=null;
          FileInputStream fe = null;
          ObjectInputStream ois = null;
          try{
                f = new File("amigos.txt");
                if (f.exists()){
                      fe = new FileInputStream(f);
                      ois = new ObjectInputStream(fe);
                      while(true){
                            amigo a = null;
```

```
                        a = (amigo)ois.readObject();
                        a.print();
                        System.out.println("");
                    }
                }
            }catch (EOFException eof) {
                System.out.println(" ---------------------------");
            }catch (FileNotFoundException fnf) {
                System.err.println("Fichero no encontrado " + fnf);
            }catch(IOException e){
                System.err.println("Se ha producido una IOException");
                e.printStackTrace();
            }catch (Throwable e) {
                System.err.println("Error de programa: " + e);
                e.printStackTrace();
            }finally{
                if (ois != null) {
                    ois.close();
                    fe.close();
                }
            }
        }catch(IOException e){
            e.printStackTrace();
        }
    }
    }
```

Se va a comentar únicamente el código resaltado:

```
public class amigo implements java.io.Serializable
```

Como se explicó anteriormente, la clase que va a ser objeto de serialización debe de ser marcada como *Serializable* mediante la sentencia anterior. Si no lo hacemos, el programa a la hora de la compilación dará error. Como se puede observar, a la clase *amigo* no hay que hacerle ninguna modificación, únicamente basta con marcarla como *Serializable*.

```
FileOutputStream fs = new FileOutputStream("amigos.txt");
ObjectOutputStream oos = new ObjectOutputStream(fs);
```

En la clase *TestFichero3*, la cual trabaja con objetos seriados, tenemos que crear un objeto **ObjectOutputStream** asociado a un objeto **FileOutputStream**, de esa forma podemos escribir objetos en un flujo de salida y, al conectarlo al fichero, estos objetos podrán ser escritos en él.

```
oos.writeObject(a);
```

El siguiente paso será escribir los objetos en el flujo de salida mediante el método writeObject del objeto de tipo *ObjectOutputStream*.

El proceso de lectura es análogo al anterior.

```
fe = new FileInputStream(f);
ois = new ObjectInputStream(fe);
while(true){
    amigo a = null;
    a = (amigo)ois.readObject();
    a.print();
    System.out.println("");
}
```

Para leer objetos de un fichero lo primero que habrá que hacer es establecer una conexión con el fichero real. El flujo con capacidad de leer información de dicho fichero (**FileInputStream**) deberá asociarse al flujo capaz de transformar esos datos en objetos (**ObjectInputStream**) para poder ser utilizados en el programa. Una vez hecho esto habrá que extraer (porque el programa así lo requiere) uno a uno los objetos para ser visualizados mediante el método *print()* de la clase *amigo*.

```
amigo a = null;
a = (amigo)ois.readObject();
```

En las líneas de código anterior se puede observar cómo se utiliza el método *readObject()* para leer el objeto del flujo de entrada. Para realizar la asignación se realiza un *casting* donde se le dice a Java entre paréntesis "**(amigo)**" que esos datos los tiene que convertir en un objeto de tipo *amigo*.

6.6 INTERFACES DE USUARIO

Cuando nos referimos a interfaz de usuario nos referimos a una interfaz gráfica. Las interfaces gráficas se caracterizan por una serie de componentes como botones, cajas de texto, etiquetas, paneles, barras de desplazamiento, etc. Java tiene principalmente dos librerías (APIs - *Application Program Interface*) bajo las cuales se pueden realizar aplicaciones con interfaz gráfica, **AWT** y **Swing**.

- **AWT** (*Abstract Window Toolkit* – Kit de herramientas abstracto para ventanas). Es parte de la JFC (*Java Foundation Classes* – Clases base de Java). El desarrollar con este kit tiene la ventaja de que las aplicaciones se parecen mucho al kit de herramientas nativo subyacente. Esto quiere decir, que cuando ejecuto el programa en Mac parece una aplicación de Mac y cuando lo ejecuto en Linux parece una aplicación Linux. Estos componentes se encuentran en la librería **java.AWT**.
- **Swing**. La ventaja de Swing frente a AWT es que los componentes utilizados por la librería gráfica de *Swing* están programados con código no nativo, lo cual lo hacen más portable. Estos componentes son más potentes que los anteriores y se identifican con una J antes del nombre del componente (por ejemplo *JButton*). Estos componentes se encuentran en la librería **javax.swing** y son todos subclases de la clase **JComponent. Swing forma parte de Java 2 y como extensión en Java 1.1**.

6.6.1 NUESTRA PRIMERA APLICACIÓN CON SWING

La primera aplicación que se realiza con cualquier lenguaje de programación es el famoso Hola Mundo. Vamos a mostrar como sería nuestra aplicación Hola Mundo y explicarla posteriormente paso a paso:

```
import javax.swing.*;
public class holamundoswing{
public static void main(String[] args) {
    JFrame frame = new JFrame("Ventana Hola Mundo");
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    JLabel label= new JLabel("Hola Mundo");
    frame.getContentPane().add(label);
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
}
}
```

Vamos a ir comentando esta aplicación paso a paso. El primer paso a realizar cuando se realiza una aplicación con *Swing* es importar la librería de *Swing* con el comando import:

```
import javax.swing.*;
```

Toda aplicación Java que utilice una interfaz con *Swing* necesita como mínimo un contenedor *Swing* de alto nivel que puede ser alguno de los siguientes:

- **JFrame**. Implementa una ventana. Las ventanas principales de una aplicación deberían de ser un *JFrame*.
- **JDialog**. Implementa una ventana tipo diálogo. Este tipo de ventanas se utilizan como ventanas secundarias y generalmente son llamadas por ventanas padre del tipo *JFrame*.
- **JApplet**. Un *applet* es una aplicación Java que se ejecuta dentro de un navegador web en la máquina del cliente. Un *JApplet* es una zona de la ventana del navegador donde se va a ejecutar el *applet*.

```
JFrame frame = new JFrame("Ventana Hola Mundo");
```

En nuestro ejemplo, el contenedor de alto nivel utilizado es el *JFrame*.

```
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

La línea anterior de código muestra lo que hará la aplicación cuando se pulse el botón de cerrar. La operación EXIT_ON_CLOSE hace que el programa termine cuando el usuario cierra la ventana. En este caso, como el programa tiene solo un *frame* el cierre del mismo terminará el programa.

```
JLabel label= new JLabel("Hola Mundo");
frame.getContentPane().add(label);
```

El siguiente paso, una vez que está creado el *frame* o ventana, es añadir componentes en él. Como se puede observar en las dos líneas de código anterior se crea un componente tipo *Label* y la añadimos al panel.

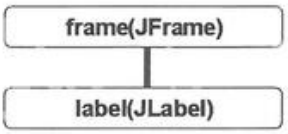


Figura 6.16. Jerarquía de la clase JFrame y JLabel

```
frame.pack();
```

El método *pack* lo que hace es establecer el tamaño del *frame*. De esta forma lo que se hace es darle el tamaño más adecuado a todos los componentes. Si lo que se quiere es establecer explícitamente el tamaño de la ventana se pueden utilizar los métodos:

- **setSize**.
- **setBounds**. Con este método se puede también establecer la posición de la ventana.

```
frame.setLocationRelativeTo(null);
```

El siguiente paso en nuestro programa es centrar la aplicación en la pantalla. Con el comando anterior se centraría la ventana creada.

```
frame.setVisible(true);
```

Por último, necesitamos hacer visible la ventana con el comando anterior. Otra forma de hacer visible la ventana es utilizar el método *show*.

6.6.2 LOS COMPONENTES SWING

Los componentes Swing son clases en sí mismos. Su utilización no difiere a la utilización de otro objeto. *Swing* provee objetos para todos los componentes básicos que se ejecutan en interfaces gráficas. Los más comunes son los siguientes:

Tabla 6.10. Componentes Swing

Objeto	Descripción
JButton	Botón estándar.
JLabel	Etiqueta de texto estándar.
JTextField	Cuadro de texto.
JTextArea	Cuadro de texto multilínea.
JCheckBox	Checkbox o casilla de verificación.
JRadioButton	Radiobutton o botones de opción.
JComboBox	Lista desplegable.
JScrollBar	Barra de desplazamiento.

En la siguiente figura se pueden apreciar algunos de los componentes arriba antes citados.

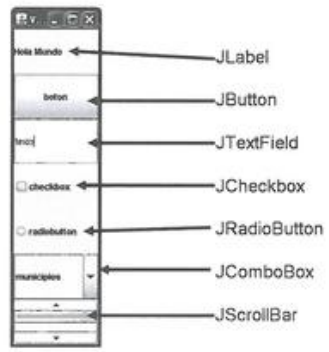


Figura 6.17. Componentes Swing

6.6.3 LOS CONTENEDORES SWING

Como antes se ha explicado, para realizar una aplicación Java necesitaremos utilizar un contenedor de nivel superior como pueden ser un **JFrame**, un **JDialog** o bien un **JApplet** en el caso de queramos que la aplicación se ejecute dentro de un navegador Web.

Existen otro tipo de contenedores intermedios como son los *JPanel*. En el programa *holamundoswing* que vimos anteriormente no los hemos utilizado dado que la aplicación era muy básica. No obstante, cuando hay que colocar una serie de controles en una ventana es imprescindible utilizar paneles para darle a la interfaz la disposición y el aspecto deseado.

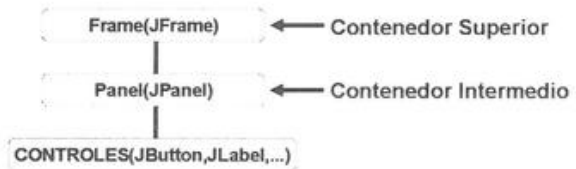


Figura 6.18. Contenedores Swing

Recuerda

Existen herramientas como Eclipse o Matisse de NetBeans que tienen un editor gráfico de interfaces y permiten diseñar ventanas de una manera sencilla y cómoda colocando los controles en la posición y orden requerido generando luego el código.

Los paneles son contenedores de componentes ligeros y estos a su vez pueden contener otros paneles. Los paneles pueden tener un color de fondo (incluso ser opacos o transparentes) o pueden cambiar la apariencia de los bordes.

El API de la clase *JPanel* es mínimo, se pueden hacer pocas operaciones con los paneles aparte de establecer bordes o formas de organización de los controles en ellos (método *setLayout*).

6.6.4 ORGANIZACIÓN DE LOS CONTROLES EN UN CONTENEDOR

Para organizar los controles en un objeto que implemente la interfaz *LayoutManager* (por ejemplo los paneles) es necesario establecer en ella un *Layout Manager* o administración de diseño. Un contenedor tiene predefinido por defecto un *Layout Manager* pero es posible e incluso deseable el cambiarlo cuando se realiza una aplicación.

Veamos algunos Layout Manager:

- **FlowLayout**. Coloca los componentes en el contenedor de izquierda a derecha. Es el *Layout Manager* por defecto en los paneles.
- **BorderLayout**. Divide el contenedor en cinco partes (norte, sur, este, oeste y centro).
- **CardLayout**. Permite colocar grupos de componentes diferentes en momentos diferentes de la ejecución del programa.
- **GridLayout**. Coloca los components en filas y columnas.
- **GridBagLayout**. Coloca los componentes en filas y columnas, pero un componente puede ocupar más de una columna.
- **BoxLayout**. Coloca los componentes en una fila o columna ajustándose.

Los *Layout Manager* se pueden establecer al crear el objeto (*constructor*):

```
Jpanel panel = new JPanel(new FlowLayout());
```

O bien una vez que el objeto ha sido creado con el método *setLayout*:

```
panel.setLayout(new FlowLayout());
```

En ocasiones, cuando se añade un componente al contenedor hay que especificar la posición que queremos que ocupe en el mismo:

```
panel.add(uncomponente, BorderLayout.PAGE_START);
```

Cuando se crea una interfaz, en ocasiones se quiere modificar el espacio entre componentes (unas veces los componentes aparecen muy apelotonados y otras muy separados). Esto se puede hacer modificando el Layout *Manager*, añadiendo componentes invisibles (no aparecen pero ocupan espacio), o bien, se puede jugar con los bordes de los componentes haciéndolos más gruesos o delgados (existen múltiples tipos de bordes aparte de su grosor).

También es posible cambiar el orden de colocación de los componentes en un contenedor.

Figura 6.19. Orden de colocación de los componentes en un contenedor

Por ejemplo, cuando se elige la ubicación de componentes según un patrón *FlowLayout*, los botones se van colocando de izquierda a derecha. No obstante, es posible modificar esta orientación mediante la siguiente sentencia (aplicada a JFrame o a JPanel):

```
frame.applyComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
```

6.6.5 APARIENCIA DE LAS VENTANAS

La apariencia (*look and feel*) genérica de las ventanas de nuestras aplicaciones Java es algo que puede ser modificado según diferentes estilos. Java tiene un gestor de la interfaz del usuario UIManager que controla la apariencia genérica de las ventanas y de los componentes que las componen. Este UIManager se encuentra en la clase javax.swing.UIManager.

Mediante la sentencia:

```
UIManager.getSystemLookAndFeelClassName()
```

Podemos recuperar el estilo del sistema. Si nuestro sistema es Windows®, al llamar al método anterior, el sistema nos devolverá lo siguiente:

```
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Dependiendo del sistema devolverá una apariencia u otra. Por ejemplo, en un sistema Solaris devolverá la apariencia Motif X-Window, que es la que trae por defecto Solaris.

Tabla 6.11. El método getSystemLookAndFeelClassName()

Plataforma	Valor devuelto por el método getSystemLookAndFeelClassName()
Multiplataforma	"javax.swing.plaf.metal.MetalLookAndFeel". Es el valor devuelto por el método getCrossPlatformLookAndFeelClassName. Este look and feel se le llama metal.
Windows®	"com.sun.java.swing.plaf.windows.WindowsLookAndFeel". Look and feel de sistemas Windows.
Solaris®	"com.sun.java.swing.plaf.motif.MotifLookAndFeel". Este look and feel puede usarse en cualquier plataforma.
Mac®	"javax.swing.plaf.mac.MacLookAndFeel". Look and feel de sistemas Mac.
Unix/Linux	"com.sun.java.swing.plaf.gtk.GTKLookAndFeel". Para sistemas GTK.

Para establecer el *look and feel* deseado en nuestro sistema podemos utilizar el siguiente código:

```
import javax.swing.UIManager;
...
//Establecer el look and feel por defecto
try{
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
}catch (Exception e){e.printStackTrace();}
//Establecer el look and feel multiplataforma
try{
UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
}catch (Exception e){e.printStackTrace();}
//Establecer el look and feel metal. Un L&F muy Java
try{
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
}catch (Exception e){e.printStackTrace();}
//Establecer el L&F Motif
try{
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
}catch (Exception e){e.printStackTrace();}
```

6.7 CONCEPTO DE EVENTO Y CONTROLADORES DE EVENTOS

En todas las aplicaciones, cuando el usuario interactúa con las mismas, suceden cosas. Si por ejemplo, pulso un botón, cierro una ventana, muevo una barra de desplazamiento, etc., el programa deberá de realizar algunas acciones. Estas acciones deberán de estar programadas, como es lógico

Figura 6.20. Evento y controlador de eventos

Existen una serie de manejadores/controladores de eventos (*listener*) los cuales deberán de ser asociados al componente para que este ejecute la respuesta necesaria. Estos *listener* son diferentes dependiendo de los eventos a los que van a dar respuesta. Es decir, los *listener* están especializados dependiendo del evento ocurrido.

En la siguiente tabla se muestra un resumen de los *listener*, los componentes y las acciones a las que responden:

Tabla 6.12. Listeners asociados a componentes

Listener	Componentes	Acción a la que responden
ActionListener	1. JButton 2. JTextField 3. JComboBox 4. ...	1. Presionar el botón. 2. Pulsar intro. 3. Elegir una opción. 4. ...
AdjustmentListener	1. JScrollBar 2. ...	Mover la barra de desplazamiento. ...
FocusListener	1. JButton 2. JTextField 3. JComboBox 4. ...	Las acciones de este listener son obtener y perder el foco (colocarnos en el componente e irnos del mismo cuando estaba activo).
ItemListener	1. JCheckbox 2. ...	1. Seleccionar y deseleccionar la opción.
KeyListener	1. JTextField 2. JTextArea 3. ...	Pulsar una tecla cuando el componente tiene el foco.
MouseListener	Múltiples componentes	Acciones como presionar el botón del ratón.
MouseMotionListener	Múltiples componnentes	Acciones como arrastrar (*drag*) o pasar por encima del objeto.
WindowListener	1. JFrame	Acciones relativas a la ventana como por ejemplo cerrarla.

La programación del manejo de eventos en una interfaz funciona de la siguiente forma:

- Se crea el componente.
- Se añade el *listener* adecuado al componente y el *listener* escuchará la acción sobre el componente.
- Dependiendo del componente o la acción se ejecutará la acción necesaria.

Figura 6.21. Ejecución de un método tras la llegada de un evento

Vamos a ver todo esto con un ejemplo concreto.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SwingTest {
    private static JLabel label = new JLabel("--");
    private static JButton btnlimpia = new JButton("Limpia");
    private static JButton btnescribe = new JButton("Escribe");
    public static void acciones(ActionEvent e) {
          Object obj=e.getSource();
          if (obj == btnlimpia){
                label.setText("");
          }
          if (obj == btnescribe){
                label.setText("Hola Mundo");
          }
    }
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }
        JFrame frame = new JFrame("Controlando eventos");
        btnlimpia.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        acciones(e);
            }
        });
        btnescribe.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        acciones(e);
            }
        });
        frame.getContentPane().add(label);
        frame.getContentPane().add(btnlimpia);
        frame.getContentPane().add(btnescribe);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.setLayout(new GridLayout(0,1));
        frame.pack();
        frame.setVisible(true);
    }
}
```

El aspecto de la aplicación creada es el siguiente:

Figura 6.22. Aspecto de la aplicación HolaMundo con eventos

Vamos a comentar las líneas de código nuevas que han ido apareciendo:

```
try {
UIManager.setLookAndFeel(
UIManager.getCrossPlatformLookAndFeelClassName());
} catch (Exception e) { }
```

En las líneas de código anteriores se establece el **aspecto general de la aplicación** el cual, como antes se indicó, es del tipo multiplataforma.

```
public static void acciones(ActionEvent e) {
    Object obj=e.getSource();
    if (obj == btnlimpia){
          label.setText("");
    }
    if (obj == btnescribe){
          label.setText("Hola Mundo");
    }
}
```

La función anterior dependiendo del tipo de botón que ha originado el evento escribirá una u otra cosa en la etiqueta de la aplicación. Mediante la función *e.getSource()* se puede **saber el objeto que generó el evento**.

```
btnlimpia.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
    acciones(e);
}
});
```

```
btnescribe.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
    acciones(e);
}
});
```

En las líneas anteriores se **crea un listener para los dos botones** de la aplicación (*btnescribe* y *btnlimpia*). Los dos *listener* ejecutarán el método *acciones*, el cual distinguirá qué botón ha generado el evento y realizará la acción correspondiente.

```
frame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
}
});
```

Para la ventana el procedimiento es el mismo que para los botones. La acción a ejecutar al cerrar la misma será la terminación de la aplicación con código 0 (todo correcto).

6.8 GENERACIÓN DE PROGRAMAS EN ENTORNO GRÁFICO

En este apartado vamos a ver un ejemplo algo más sofisticado de aplicación. La idea es crear una aplicación que haga la conversión de euros a dólares y viceversa. Para realizar dicha aplicación deberemos crear un *frame* que contenga tres paneles y en cada uno de ellos una etiqueta (*label*) y un campo de texto (*textfield*). En el primero y último habrá una barra de desplazamiento (*slider*) que mostrará las cifras del campo de texto de una manera más visual.

El esquema visual de la aplicación es el siguiente:

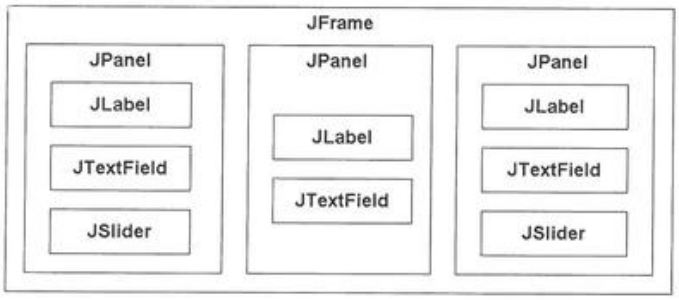


Figura 6.23. Esquema de componentes de una aplicación

Una vez definidos los contenedores y componentes, la aplicación tendrá el siguiente aspecto:

Figura 6.24. Aspecto de la aplicación Conversor Euros-Dólares

Curioso
Como te habrás dado cuenta, la aplicación de la figura anterior no está del todo terminada (faltan acentos, el cambio se podría introducir con una coma en vez de con punto, los slider podrían habilitarse y ser más funcionales, etc.). Estos cambios se proponen al alumno como ejercicios al final del tema.

El código de la aplicación sería el siguiente:

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

public class SwingConversor {
    static final int MIN = 0;
    static final int MAX = 1000;
    static final int INIT = 0;
    private static JLabel label = new JLabel("1 Euro son en dolares:");
    private static JLabel lbleuros = new JLabel("Euros:");
    private static JLabel lbldolares = new JLabel("Dolares:");
    private static JFrame frame = new JFrame("Conversor Euros - Dolares");
    private static JPanel panel1 = new JPanel();
    private static JPanel panel2 = new JPanel();
    private static JPanel panel3 = new JPanel();
    private static JTextField txteuro = new JTextField("0");
    private static JTextField txtdolar = new JTextField("0");
    private static JTextField txtcambio = new JTextField("1.36");
```

```
private static JSlider sliderdolar = new JSlider(JSlider.HORIZONTAL,MIN,MAX,INIT);
private static JSlider slidereuro = new JSlider(JSlider.HORIZONTAL,MIN,MAX,INIT);
public static void cambiotexto(ActionEvent e) {
      if ( e.getSource() == txteuro ){
            float icambio=Float.parseFloat(txteuro.getText());
            icambio=100*icambio*Float.parseFloat(txtcambio.getText());
            icambio = Math.round(icambio);
            icambio = icambio/100;
       txtdolar.setText(String.valueOf(icambio));
       //cambiar los slider
       sliderdolar.setValue(Math.round(Float.parseFloat(txtdolar.getText())));
       slidereuro.setValue(Math.round(Float.parseFloat(txteuro.getText())));
      }
      if ( e.getSource() == txtdolar ){
            System.out.println("dentro");
            float icambio=Float.parseFloat(txtdolar.getText());
            icambio=100*icambio/Float.parseFloat(txtcambio.getText());
            icambio = Math.round(icambio);
            icambio = icambio/100;
       txteuro.setText(String.valueOf(icambio));
      }
}
public static void mueveSlider(ChangeEvent e) {
      int valor;
      JSlider obj=(JSlider)e.getSource();
      System.out.println(obj.getValueIsAdjusting());
      System.out.println(obj.getValue());

      if (!obj.getValueIsAdjusting()) {
            System.out.println(obj.getValue());
            valor = (int)obj.getValue();
            if (obj == sliderdolar){
                  txtdolar.setText(String.valueOf(valor));
                  float icambio=100*valor/Float.parseFloat(txtcambio.getText());
                  icambio=Math.round(icambio);
                  icambio=icambio/100;
                  //cambiar el txteuro
                  txteuro.setText(String.valueOf(icambio));
                  //cambiar el slidereuro
                  int i = Math.round(icambio);
                  slidereuro.setValue(i);
            }
            if (obj == slidereuro){
                  txteuro.setText(String.valueOf(valor));
```

```
                        float icambio=100*valor*Float.parseFloat(txtcambio.getText());
                        icambio=Math.round(icambio);
                        icambio=icambio/100;
                        //cambiar el txtdolar
                        txtdolar.setText(String.valueOf(icambio));
                        //cambiar el sliderdolar
                        int i = Math.round(icambio);
                        slidereuro.setValue(i);
                }
        }
    }
    public static void colocaelementos(){
            frame.getContentPane().add(panel1);
            frame.getContentPane().add(panel2);
            frame.getContentPane().add(panel3);

            slidereuro.setBorder(BorderFactory.createTitledBorder("Euros"));
            slidereuro.setMajorTickSpacing(200);
            slidereuro.setMinorTickSpacing(100);
            slidereuro.setPaintTicks(true);
            slidereuro.setPaintLabels(true);
            slidereuro.disable();
            sliderdolar.setBorder(BorderFactory.createTitledBorder("Dolares"));
            sliderdolar.setMajorTickSpacing(200);
            sliderdolar.setMinorTickSpacing(100);
            sliderdolar.setPaintTicks(true);
            sliderdolar.setPaintLabels(true);
            sliderdolar.disable();
            panel1.add(lbleuros);
            panel1.add(txteuro);
            panel1.add(slidereuro);

            panel2.add(label);
            panel2.add(txtcambio);

            panel3.add(lbldolares);
            panel3.add(txtdolar);
            panel3.add(sliderdolar);
frame.addWindowListener(new WindowAdapter() {
                            public void windowClosing(WindowEvent e) {
                        System.exit(0);
                            }
                });
txteuro.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                        cambiotexto(e);
            }
});
txtdolar.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        cambiotexto(e);
            }
});

frame.setLayout(new FlowLayout());
panel1.setLayout(new GridLayout(0,1));
panel2.setLayout(new GridLayout(0,1));
panel3.setLayout(new GridLayout(0,1));
frame.pack();
frame.setVisible(true);
    }
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }
        colocaelementos();
    }
}
```

RESUMEN DEL CAPÍTULO

En este capítulo se estudia la persistencia de información en ficheros. Los ficheros junto con las bases de datos son el almacen de información por excelencia. Una vez comprendido el concepto de almacenamiento y realizados los ejercicios del tema le resultará más fácil comprender el almacenamiento en bases de datos que se verá en el capítulo 8. Otra parte muy importante del capítulo es el desarrollo de interfaces de usuario. Actualmente, cualquier aplicación utiliza interfaces de usuario y, por lo tanto, es un elemento imprescindible en el aprendizaje del lenguaje Java.

EJERCICIOS RESUELTOS

1. Necesitamos crear una clase cuenta con un método *cuentaPalabras()* que cuente las palabras existentes en un archivo de datos pasado como parámetro.

Solución:

```
import java.io.*;
import java.util.StringTokenizer;
public class cuenta{
    public int cuentaPalabras(String fichero){
        int contador=0;
        try{
            File fe = new File(fichero);
            FileReader fr = new FileReader(fe);
            BufferedReader br = new BufferedReader(fr);
            String s;
            while((s = br.readLine()) != null) {
                StringTokenizer str;
                str = new StringTokenizer(s);
                contador += str.countTokens();
            }
            if (fr != null) fr.close();
        }catch (FileNotFoundException fnf) {
                System.err.println("Fichero no encontrado " + fichero);
        }catch(IOException e){
                System.err.println("Se ha producido una IOException");
                e.printStackTrace();
        }catch (Throwable e) {
                System.err.println("Error de programa: " + e);
                e.printStackTrace();
        }
        return contador;
    }
    public static void main(String[] args){
        cuenta c = new cuenta();
        int d=c.cuentaPalabras("datos.txt");
        System.out.println(d);
    }
}
```

2. Se necesita crear una clase censura con un método aplicaCensura que modifique ciertas palabras de un fichero. El método toma un fichero de entrada y mediante un fichero de censura creará un fichero de salida con la modificaciones necesarias.

Ejemplo:

Fichero de entrada

En un lugar de la mancha de cuyo nombre no quiero acordarme...

vivia un hidalgo de adarga estrecha

Fichero censura

acordarme recordar

hidalgo noble

Fichero de salida

En un lugar de la mancha de cuyo nombre no quiero recordar...

vivia un noble de adarga estrecha

Solución:

```
import java.io.*;
import java.util.StringTokenizer;
public class censura{
    public void aplicaCensura(String fentrada,String fcensura, String fsalida){
          try{
                File fe = new File(fentrada);
                FileReader fr = new FileReader(fe);
                BufferedReader br = new BufferedReader(fr);

                File fs = new File(fsalida);
                FileWriter fw = new FileWriter(fs);

                String s;
                while((s = br.readLine()) != null) {
                      File fc = new File(fcensura);
                      FileReader frc = new FileReader(fc);
                      BufferedReader brc = new BufferedReader(frc);

                      String scen;
                      while ((scen = brc.readLine()) != null) {
                            StringTokenizer str;
                            str = new StringTokenizer(scen);
                            s=s.replace(str.nextToken(),str.nextToken());
                      }
                      System.out.println(s);
```

```
                    fw.write(s);
                    fw.write("\r\n");

                    if (frc != null) frc.close();
                }
                if (fw != null) fw.close();
                if (fr != null) fr.close();
            }catch (FileNotFoundException fnf) {
                    System.err.println("Fichero no encontrado ");
            }catch(IOException e){
                    System.err.println("Se ha producido una IOException");
                    e.printStackTrace();
            }catch (Throwable e) {
                    System.err.println("Error de programa: " + e);
                    e.printStackTrace();
            }
    }
    public static void main(String[] args){
            cuenta c = new cuenta();
            c.aplicaCensura("datos.txt","censura.txt","salida.txt");
    }
}
```

Modifica este programa para que el fichero de salida sea el mismo fichero de entrada.

3. Tenemos un fichero con una serie de números los cuales queremos ordenar de manera ascendente. El objetivo es que los números queden ordenados en el mismo fichero. Para la resolución del problema crea una clase orden con un método ordena que haga la ordenación de los datos del fichero. Los números están cada uno en una línea del fichero y el fichero cuenta con al menos 5 números.

Solución:

La solución que se le ha dado al problema se ubica en el método *ordenar* y consta de tres fases:

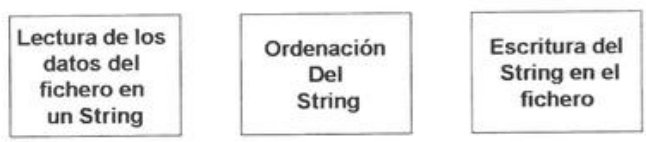


Figura 6.25. *Pasos en la resolución del problema*

```
import java.io.*;
import java.util.StringTokenizer;
public class cuenta{
    public void ordenar(String fichero){
            try{
```

```
                File fe = new File(fichero);
                FileReader fr = new FileReader(fe);
                BufferedReader br = new BufferedReader(fr);

                String numeros=new String();
                String s;
                while((s = br.readLine()) != null) {
                        numeros += s + " ";
                }
                System.out.println(numeros);

                if (fr != null) fr.close();

                StringTokenizer str;
                boolean ordenado = false;
                while (!ordenado){
                        ordenado = true;
                        String anterior, posterior="";
                        str = new StringTokenizer(numeros);
                        anterior = str.nextToken();
                        numeros = "";
                        while (str.hasMoreTokens()) {
                                posterior=str.nextToken();
                                if      (Integer.parseInt(anterior)>Integer.
parseInt(posterior)){
                                        String aux = anterior;
                                        anterior = posterior;
                                        posterior = aux;
                                        ordenado = false;
                                }
                                numeros += anterior + " ";
                                anterior = posterior;
                        }
                        numeros += posterior;

                        System.out.println(numeros);
                }

                File fs = new File(fichero);
                FileWriter fw = new FileWriter(fs);

                str = new StringTokenizer(numeros);
                while (str.hasMoreTokens()) {
                        fw.write(str.nextToken());
                        fw.write("\r\n");
                }

                if (fw != null) fw.close();
```

```
            }catch (FileNotFoundException fnf) {
                        System.err.println("Fichero no encontrado ");
            }catch(IOException e){
                        System.err.println("Se ha producido una IOException");
                        e.printStackTrace();
            }catch (Throwable e) {
                        System.err.println("Error de programa: " + e);
                        e.printStackTrace();
            }
      }
      public static void main(String[] args){
            c.ordenar("numeros.txt");
      }
}
```

Pasamos a comentar la parte de la ordenacion que es la parte con más complejidad:

```
StringTokenizer str;
boolean ordenado = false;
while (!ordenado){
```

El objetivo es ordenar el *String*, para eso utilizamos la clase *tokenizer* para desmenuzarlo e ir extrayendo número a número (*token* a *token*). El *String* se supone que está ordenado, en caso contrario ponemos la variable ordenado a false. Si esta variable no cambia el *String* estará ordenado y el bucle termina.

```
      ordenado = true;
      String anterior, posterior="";
      str = new StringTokenizer(numeros);
      anterior = str.nextToken();
      numeros = "";
```

Se utilizan dos variables, anterior y posterior. Estas variables van a contener el número anterior y posterior extraido del *String*.

```
      while (str.hasMoreTokens()) {
            posterior=str.nextToken();
            if (Integer.parseInt(anterior)>Integer.
parseInt(posterior)){
                  String aux = anterior;
                  anterior = posterior;
                  posterior = aux;
                  ordenado = false;
            }
```

Se recorre el *String* y si el número anterior es mayor al posterior se ordenan. En el caso de que haya que ordenar se pone la variable ordenado a *false* pues no se sabe si todo el *String* está ya ordenado.

```
numeros += anterior + " ";
anterior = posterior;
}
numeros += posterior;
```

Se van concatenando los datos ordenados en el mismo *String* números. La siguiente línea de código muestra el resultado de la pasada.

```
System.out.println(numeros);
}
```

- Modifica este programa para que ordene el fichero aunque solamente tenga un número.

EJERCICIOS PROPUESTOS

- **1.** La aplicación de cambio de euros a dólares necesita unos retoques como la colocación de acentos en los literales (en la palabra dólares). Modifícala para que los literales aparezcan con acento.

- **2.** Modifica la aplicación de conversión de euros a dólares para que cuando el usuario mueva los *sliders*, los valores queden reflejados en el campo de texto correspondiente y haga la conversión directamente.

- **3.** Modifica la aplicación de conversión de euros a dólares para que cuando el usuario modifique el cambio euro-dólar de igual que lo introduzca con punto o con coma.

- **4.** Realiza un programa que genere primitivas de forma aleatoria. El programa tendrá que tener la siguiente apariencia:

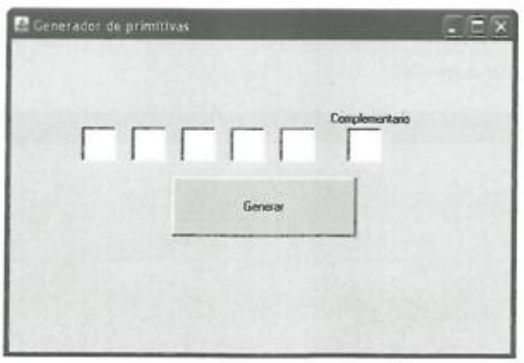


***Figura 6.26.** Aspecto de la aplicación generadora de primitivas*

Ten en cuenta en la generación de la primitiva que los números no podrán repetirse en la misma primitiva.

5. Realiza un programa que compruebe si el usuario conoce los componentes de tres platos. El programa deberá tener la siguiente apariencia:

Figura 6.27. Aspecto de la aplicación Cheff 4000

El programa dará correcto si tras comprobar el usuario ha elegido lo siguiente:

Tabla 6.13. Tabla ejercicio 5

Cocido	Lentejas	Judías
Chorizo	Lentejas	Judías
Morcilla	Chorizo	Nuez moscada
Garbanzos	Cebolla	Laurel
Zanahoria	Zanahoria	Colorante
	Pimentón	Chorizo
		Cebolla

6. (Ejercicio de dificultad alta) Realiza un programa que traduzca palabras del español al inglés y viceversa. El programa deberá tener el siguiente aspecto:

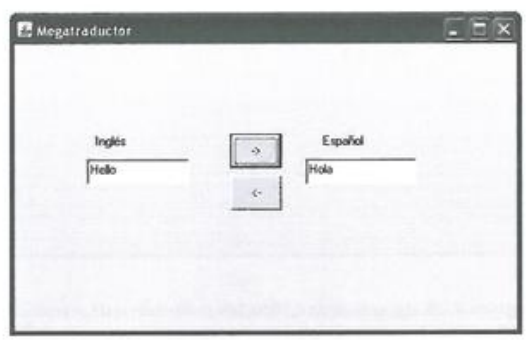


Figura 6.28. Aspecto de la aplicación Megatraductor

Para almacenar el diccionario de datos el alumno deberá de utilizar ficheros. Se deja a elección del alumno el formato del fichero de datos.

- **7.** Realiza una aplicación que haga la conversión de decimal a romano y viceversa. El aspecto de la interfaz deberá de ser parecido al siguiente:

Figura 6.29. Aspecto de la aplicación Conversor Romano - Decimal

- **8.** Realiza un programa que almacene y recupere un objeto persona en un fichero. La clase *persona* contiene los siguientes atributos:
  - Nombre. Campo alfanumérico.
  - Apellidos. Campo alfanumérico.
  - Teléfono. Campo numérico.

7 Estructuras de almacenamiento

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las diferencias entre el almacenamiento en memoria y el persistente en disco ya visto.
- ✓ Comprender la importancia y utilidad de las estructuras de almacenamiento en memoria.
- ✓ Resolver problemas utilizando arrays.
- ✓ Comprender y saber aplicar los algoritmos de ordenación.
- ✓ Estudiar los vectores de objetos.

7.1 ARRAYS O VECTORES

Hasta ahora, en todos los ejercicios que se han ido viendo durante los temas anteriores no había una necesidad muy grande de almacenar muchos valores, se utilizaban variables para almacenar 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, nada más que 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 se llama 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.

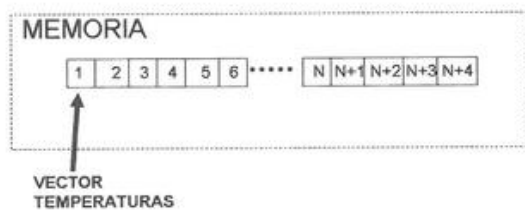


Figura 7.1. Vector temperaturas

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.

7.1.1 DECLARACIÓN DE VECTORES

En Java se pueden declarar vectores de dos formas diferentes. Para declarar nuestro vector de temperaturas se puede realizar 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.

7.1.2 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 puede asignársele 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 línea del código anterior en una sola.

7.1.3 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 código anterior se ha creado un vector de 12 posiciones del tipo byte con los valores especificados.

7.1.4 MÉTODOS DE LOS VECTORES

Como se ha dicho anteriormente, Java maneja los vectores como si fueran objetos, por lo tanto existen una serie de 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* si da como resultado *true*.

7.1.5 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.

7.2 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 código anterior se ha creado un array de 2 filas y tres 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 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 7.2. 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;
      }
}
```

A FONDO

LOS FICHEROS JAR

De momento los programas o aplicaciones que hemos desarrollado a lo largo de este libro son bastante sencillas, uno o varios ficheros de clases con algún fichero de datos si cabe. No obstante, las aplicaciones más profesionales suelen contener múltiples ficheros, de ahí que para distribuirlas se haga en un fichero JAR. Este formato permite empaquetar múltiples ficheros (clases, datos, sonido, imágenes, etc.) en un solo archivo y tiene muchas ventajas. Dado que están comprimidos, se pueden descargar las aplicaciones de una sola vez, proporciona mecanismos de seguridad y la portabilidad que ofrece al ser un estándar muy común de la plataforma Java.

Las posibilidades de la herramienta son muchas más de las que vamos a ver en esta sección. No obstante, aquí se van a repasar todos los pasos desde crear un JAR de la nada hasta ejecutar una aplicación contenida en un fichero JAR. Obviamente este formato incluye muchas funcionalidades, entre otras la firma electrónica mediante la cual podemos firmar nuestros ficheros JAR y demostrar ante un tercero que el contenido del mismo pertenece a nosotros y no a otra persona que intente suplantarnos.

Las características básicas que vamos a ver son las siguientes:

1. Crear un fichero JAR.
2. Ver el contenido del JAR.
3. Extraer los ficheros de un JAR.
4. Ejecutar la aplicación contenida en un JAR.

Crear un fichero JAR

Para crear el fichero JAR utilizaremos los parámetros *cf* de la herramienta JAR. Para mostrar más información hemos añadido la opción *v*.

```
C:\Archivos de programa\Geany>jar cfv hola.jar holamundoswing.class
manifest agregado
agregando: holamundoswing.class (entrada = 1450) (salida = 882) (desinflado
39%)
```

Ver el contenido del JAR

El contenido del JAR se muestra utilizando los parámetros *tf*.

```
C:\Archivos de programa\Geany>jar tfv hola.jar holamundoswing.class
1450 Tue Jan 04 17:59:10 CET 2011 holamundoswing.class
```

Extraer los ficheros de un JAR

Se pueden extraer todos o alguno de los ficheros contenidos en un JAR. El siguiente comando extrae todos los ficheros del JAR:

```
C:\Archivos de programa\Geany>jar xfv hola.jar
 creado: META-INF/
 inflado: META-INF/MANIFEST.MF
 inflado: holamundoswing.class
```

Y el siguiente comando extraerá solamente el archivo *holamundoswing.class*:

```
C:\Archivos de programa\Geany>jar xfv hola.jar holamundoswing.class
inflado: holamundoswing.class
```

Ejecutar la aplicación contenida en un JAR

Para ejecutar la aplicación contenida en un JAR utilizaremos los parámetros *cp*.

```
C:\Archivos de programa\Geany>java -cp hola.jar holamundoswing
(en algunas ocasiones se utiliza jre -cp en vez de java -cp)
```

La siguiente tabla muestra un resumen de los comandos vistos de la herramienta JAR.

Tabla 7.1. Comando JAR

Comando	Descripción
jar cf fichero_jar fichero/s	Crear un fichero JAR.
jar tf fichero_jar	Ver el contenido de un fichero JAR.
jar xf fichero_jar	Extraer el contenido de un fichero JAR.
jar xf fichero_jar fichero/s	Extraer ficheros de un fichero JAR.
java -cp fichero_jar.jar Clase_main o jre -cp fichero_jar.jar Clase_main	Ejecutar una aplicación empaquetada en un fichero JAR.

¿Qué es el manifest o manifiesto de un JAR?

El manifiesto contiene información sobre los ficheros contenidos en el fichero JAR y es un fichero especial. Este fichero puede adaptarse para utilizar los ficheros JAR para múltiples propósitos.

El manifiesto del JAR creado anteriormente es muy simple, tan solo contendría los siguientes datos:

```
Manifest-Version: 1.0
Created-By: 1.6.0_19 (Sun Microsystems Inc.)
```

Como se puede observar contiene la versión de Java con el que ha sido creado y poco más. En el caso de que utilicemos funcionalidades más avanzadas de la herramienta JAR el contenido de este fichero cambiaría sustancialmente.

7.3 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 código 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.

7.3.1 LA CLASE STRING

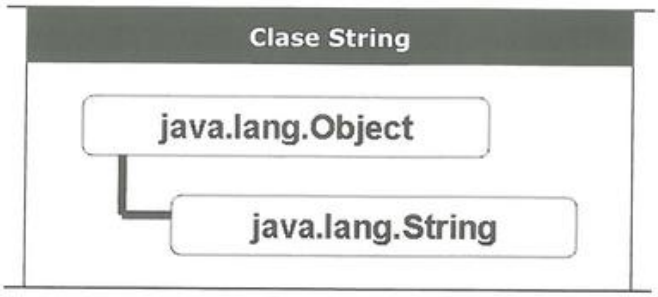


Figura 7.3. Jerarquía de la clase String

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("Lionel");
String cad3 = new String(cad2);
```

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

- **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 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".

A FONDO

DIFERENCIA ENTRE EL MÉTODO EQUALS Y EL OPERADOR ==

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

El código anterior mostrará por pantalla las siguientes cadenas: "SON IGUALES", "SON DIFERENTES". En el primer caso los dos objetos son iguales porque contienen la misma secuencia de caracteres y el segundo de los casos son diferentes porque son **diferentes objetos**. Para que en el segundo caso muestre la cadena "SON IGUALES" debería de cambiarse la línea de código:

```
String cad2=new String("EMMA");
```

Por la siguiente otra:

```
String cad2=cad1;
```

En este último caso es importante recalcar que *cad2* y *cad1* apuntan al mismo objeto.

- **String trim()**. Elimina los espacios en blanco que contenga el objeto *String* al principio y final del mismo.

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

El código anterior mostrará por pantalla la cadena "JUAN SERRANO".

- **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)** o **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);
```

A FONDO

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();
```

7.3.2 LA CLASE STRINGBUFFER

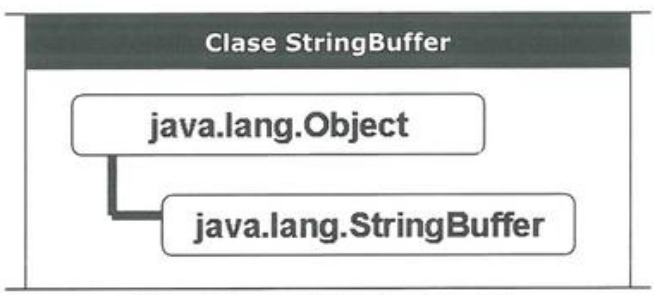


Figura 7.4. Jerarquía de la clase StringBuffer

Recuerda
Los objetos de la clase *String NO* son modificables sino que los métodos que actúan sobre los objetos devuelven un objeto nuevo con las modificaciones realizadas. En cambio, los objetos StringBuffer SÍ son modificables.

- **StringBuffer([arg])**. Constructor de la clase *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
StringBuffer apellidos = new StringBuffer(80);
StringBuffer direccion = new StringBuffer();
```

La segunda línea de código crea un objeto vacío con una capacidad para 80 caracteres. En la tercera línea de código no se especifica la capacidad pero por defecto la deja a 16.

- **int length()**. Muestra la longitud del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
System.out.println(nombre.length());
```

La salida por pantalla del código anterior será 4.

- **int capacity()**. Muestra la capacidad del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
System.out.println(nombre.capacity());
```

Curioso: La salida por pantalla del código anterior será 20 aunque su longitud era solo de 4. Eso es debido a que cuando se crea un objeto Java le otorga una capacidad igual al número de caracteres almacenados más 16.

- **StringBuffer append(argumento)**. Añade el argumento al final de la cadena de caracteres *StringBufffer*. El tipo del argumento puede ser *int*, *long*, *float*, *double*, *boolean*, *char*, *char*[ ], *String* y *Object*.

```
StringBuffer nombre = new StringBuffer("Juan Carlos");
String apellidos=new String(" Moreno Pérez");
nombre.append(apellidos);
System.out.println(nombre);
```

El código anterior muestra por pantalla la cadena "Juan Carlos Moreno Pérez".

- **StringBuffer insert(int pos, arg)**. Añade el argumento en la posición pos de la cadena de caracteres StringBufffer. El tipo del argumento puede ser *int*, *long*, *float*, *double*, *boolean*, *char*, *char*[], *String* y *Object*.

```
StringBuffer nombre = new StringBuffer("EMMA");
String apellidos=new String(" MORENO");
nombre.insert(nombre.length(),apellidos);
System.out.println(nombre);
```

El código anterior muestra por pantalla la cadena "EMMA MORENO".

- **StringBuffer reverse()**. Invierte la cadena de caracteres que contiene.

```
StringBuffer nombre = new StringBuffer("TURRION");
nombre.reverse();
System.out.println(nombre);
```

El código anterior mostrara por pantalla la cadena "NOIRRUT".

- **StringBuffer delete(int x, int y)**. Elimina los caracteres entre las posiciones *x* e *y* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("RAUL JESUS TURRION");
nombre = nombre.delete(4,10);
System.out.println(nombre);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION"

- **StringBuffer replace(int x, int y, String s)**. Reemplaza los caracteres entre las posiciones *x* e *y* por el *String s* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("RAUL JESUS");
nombre = nombre.replace(5,10,"TURRION");
System.out.println(nombre);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION".

- **String substring(int x, int y)**. Devuelve un *String* que contiene la cadena que comienza en el carácter *x* hasta el carácter *y-1* (o hasta el final si no se especifica el argumento *y*).

```
StringBuffer nombre = new StringBuffer("RAUL JESUS TURRION");
String turri = nombre.substring(0,4)+nombre.substring(10);
System.out.println(turri);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION".

- **String toString()**. Devuelve un objeto *String* el cual es una copia del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("TURRION");
String turri = nombre.toString();
System.out.println(turri);
```

El código anterior crea un objeto *String* a partir de un objeto *StringBuffer* y muestra su contenido por pantalla.

- **char charAt(int x)**. Devuelve el carácter que está en la posición *x* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("EMMA");
System.out.println(nombre.charAt(0));
```

El código anterior mostrará por pantalla el carácter 'E'.

- **void setCharAt(int x, char c)**. Reemplaza el carácter que está en la posición *x* del objeto *StringBuffer* por el carácter *c*.

```
StringBuffer nombre = new StringBuffer("EMMA");
nombre.setCharAt(0,'e');
System.out.println(nombre.toString());
```

El código anterior mostrará por pantalla la cadena de caracteres "eMMA".

A FONDO

CLASE STRINGTOKENIZER

Esta clase permite dividir una cadena de caracteres en elementos independientes si estos están separados por un espacio en blanco, un retorno de carro (\r) o de línea (\n), un avance de página (\f) o un tabulador (\t).

Un ejemplo de utilización de esta clase es el siguiente:

```
StringTokenizer str;
str = new StringTokenizer("UNO DOS TRES PERICO JUANICO Y_ANDRES");
System.out.println("La cadena str tiene "+str.countTokens()+" elementos");
while (str.hasMoreTokens()) System.out.println(str.nextToken());
```

El código anterior mostrará que la cadena str tiene 6 elementos y los irá sacando por pantalla uno a uno.

Para utilizar esta clase se debe de importar la clase *StringTokenizer*:

```
import java.util.StringTokenizer;
```

O importar todas las clases del paquete java.util:

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

También es posible especificar los delimitadores dentro del constructor de la clase:

```
str = new StringTokenizer("UNO|DOS|TRES PERICO|JUANICO|Y_ANDRES","|");
```

En el ejemplo anterior se utiliza el símbolo '|' como delimitador.

7.4 ARRAYS O VECTORES DE OBJETOS STRING

Dado que ya se conoce cómo funcionan los vectores y los objetos *String*, en este apartado se va a ver un ejemplo de utilización de estos tipos de datos. En el ejemplo se va a realizar un programa que lee nombres por teclado y los va almacenando en un vector. Una vez creado el vector se mostrarán dichos nombres por teclado según la posición en que se han leído.

```
public class test {
    private static String[]  lista;
    final static int POS=10; //número de posiciones del array
    public static void muestra(){
```

```
        for (int i=0;i<POS;i++)    System.out.print(lista[i]+" ");
    }
    public static void main(String[] args) {
        lista = new String[POS];
        for (int i=0;i<POS;i++){
            String ln = System.console().readLine();
            lista[i]=ln.toString();
        }
        System.out.println("");
        muestra();
        System.out.println("");
    }
}
```

7.5 ALGORITMOS DE ORDENACIÓN

Los algoritmos de ordenación se aplican generalmente en arrays unidimensionales (también en ficheros) y su finalidad es organizar los datos en dichos arrays. Estos algoritmos de ordenación tienen gran importancia, con lo cual en muchos lenguajes de programación existen librerías que contienen funciones o procedimientos para ordenar arrays. No todos los algoritmos ordenan de la misma forma ni todos son igual de eficientes, algunos son muy rápidos cuando el vector está casi ordenado, otros lo son cuando está muy desordenado y a otros el preordenamiento no les afecta en gran medida. Dependiendo de la estrategia, algunos algoritmos son mejores que otros. Los algoritmos se distinguen por su complejidad computacional la cual clasifica los algoritmos dependiendo de su eficiencia, en esta clasificación se tiene en cuenta el peor caso, el caso promedio y el mejor de los casos.

Tabla 7.2. Algoritmos de ordenación

Algoritmo	Estrategia de ordenación
Burbuja o bubblesort	Este ordenamiento pega una serie de pasadas al vector y compara parejas de elementos adyacentes y los intercambia si no están ordenados. En la primera pasada obviamente el mayor elemento se situará el último y así sucesivamente. Cuando en una pasada no hace ningún intercambio quiere decir que el vector ya está ordenado.
Cocktail sort	El un ordenamiento por burbuja bidireccional.

Ordenación por inserción o insertion sort	La ordenación se realiza insertando los datos ordenadamente en un vector. Los datos se van insertando agrupados y cuando se inserta un nuevo dato se le hace hueco en la posición que le corresponde desplazando los demás elementos.
Por Mezcla o Merge Sort	Es un algoritmo desarrollado por Von Newmann que emplea la técnica divide y vencerás. El algoritmo va dividiendo por la mitad de forma recursiva el vector a ordenar en dos listas las cuales deberán de ser también ordenadas. El algoritmo para de dividir cuando se queda con 0 ó 1 elementos, entonces se consideran que están ya ordenados. Si se ordenan dos listas las cuales a su vez están ordenadas, el resultado será una lista ordenada.
Ordenamiento por selección o selection sort	El funcionamiento es simple, se busca el menor elemento de la lista y se coloca en el primer lugar, luego se busca el segundo y se coloca en el segundo lugar y así sucesivamente.
Ordenación rápida o Quicksort	Es el algoritmo más rápido. Utiliza la técnica divide y vencerás. Es un algoritmo recursivo que utiliza un pivote como elementos central y coloca todos los elementos menores a su izquierda y los mayores a su derecha. La lista se separa en dos sublistas y se repite el proceso de manera recursiva con las dos sublistas hasta que toda la lista se queda ordenada.

7.5.1 ORDENACIÓN POR EL MÉTODO DE LA BURBUJA

El ordenamiento por el método de la burbuja es muy utilizado, sobre todo porque es sencillo de entender e implementar. En la siguiente tabla se puede comprender claramente cómo funciona el método de la burbuja. Como se puede apreciar, al igual que las burbujas de aire en el agua suben a la superficie, los elementos mayores de la lista se irán colocando al final del vector de forma ordenada.

1ª Pasada

4	2	9	3	1
2	4	9	3	1
2	4	3	9	1
2	4	3	1	9

2ª Pasada

2	4	3	1	9
2	3	4	1	9
2	3	1	4	9

3ª Pasada

2	3	1	4	9
2	1	3	4	9

4ª Pasada

2	1	3	4	9
1	2	3	4	9

Como parece lógico, las pasadas cada vez serán más cortas dado que en cada pasada, al menos 1 elemento quedará ordenado al final de la lista. Una implementación de este código en Java sería el siguiente:

```
public static void burbuja(int array[]){
        int aux;
        for (int i=array.length;i>0;i--){
                for (int j=0;j<i-1;j++){
                        if (array[j]>array[j+1]){
                                aux = array[j+1];
                                array[j+1]=array[j];
                                array[j]=aux;
                        }
                }
        }
}
```

Este código no es del todo eficiente porque si el vector ya está ordenado el procedimiento seguirá comprobando hasta terminar sin tener en cuenta este hecho. Para conseguir esto, basta con utilizar un *flag* que marque si el vector está ordenado o no y en el caso de que el algoritmo de una pasada sin realizar ningún intercambio de elementos el proceso parará.

A FONDO

BÚSQUEDA DE DATOS DENTRO DE UN ARRAY

Las ventajas de los arrays o vectores son la versatilidad y posibilidades que ofrecen. Una de las operaciones más frecuentes que se realizan en los mismos es la búsqueda de información dentro de ellos. La forma más fácil de buscar información en un array es realizar una búsqueda secuencial hasta encontrar el dato, pero no es la más eficiente. Imaginemos un array muy grande. De media tendremos que buscar en N/2 elementos para encontrar nuestro dato. Si el array es muy grande perderemos mucho tiempo durante la búsqueda. Un método más eficiente es la **búsqueda binaria**. En la búsqueda binaria se sigue el lema "divide y vencerás". En la siguiente imagen se puede observar cómo funciona el algoritmo:

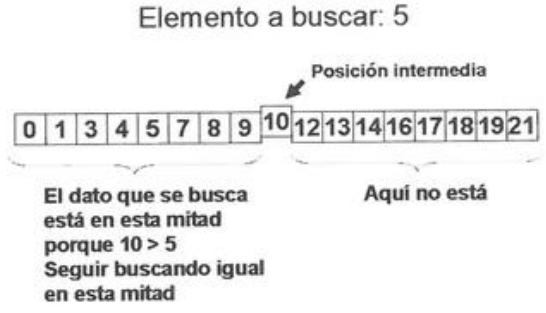


Figura 7.5. Búsqueda binaria

Importante: Los datos del array donde se va a realizar la búsqueda deberán de estar ordenados.

Como se puede ver, lo que se hace es elegir una posición intermedia y desechar de la búsqueda la mitad del array donde el dato no va a estar siguiendo la búsqueda en el lado restante.

La busqueda seguirá igual pero para la mitad del array donde debería de estar el elemento a buscar.

7.5.2 ORDENACIÓN POR EL MÉTODO DE INSERCIÓN DIRECTA

Este algoritmo funciona de la siguiente manera, se desean insertar en un array de 5 elementos los siguientes datos (4, 2, 9, 3 y1).

1ª Inserción (4)

4				

2ª Inserción (2)

4				
	4			
2	4			

3ª Inserción (9)

2	4			
2	4	**9**		

4ª Inserción (3)

2	4	9		
2		4	9	
2	**3**	4	9	

5ª Inserción (1)

2	3	4	9	
	2	3	4	9
1	2	3	4	9

A FONDO

CLASE HASHTABLE

Esta clase se encuentra en el paquete java.util. Una *Hashtable* implementa una tabla *hash* la cual crea una especie de diccionario que relaciona claves con valores.

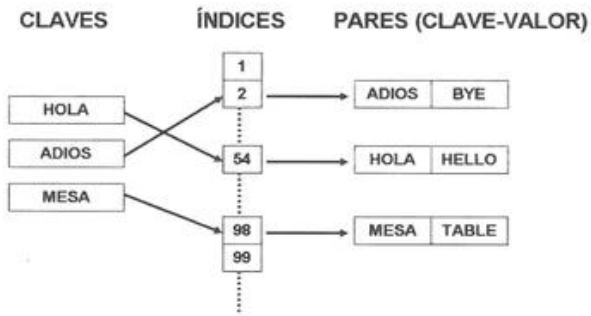


Figura 7.6. La clase Hashtable

En el siguiente ejemplo comentado se muestra cómo se crea y se trabaja con una *hashtable* como la de la figura anterior:

```
import java.util.*;
public class diccionario {
    public static void main(String[] args) {
    Hashtable dic  = new Hashtable();
    dic.put("HOLA","HELLO");
    dic.put("ADIOS", "BYE");
    dic.put("MESA","TABLE");
    dic.put("SILLA","CHAIR");
    dic.put("CABEZA","HEAD");
    dic.put("CARA","FACE");
    String saludo  = (String) dic.get("HOLA");
    String despedida  = (String) dic.get("ADIOS");
    String brazo = (String) dic.get("BRAZO");
    System.out.println("HOLA  : " + saludo);//muestra HELLO por pantalla
    System.out.println("ADIOS : " + despedida);//muestra BYE por pantalla
    System.out.println("BRAZO : " + brazo); //muestra null por pantalla
    System.out.println("dic contiene " + dic.size() + " pares.");
    if( dic.containsKey("HOLA") ){
    System.out.println("dic contiene HOLA como clave");
    }else{
    System.out.println("dic NO contiene HOLA como clave");
    }
    if( dic.contains("HELLO") ){
    System.out.println("dic contiene HELLO como valor");
    }else{
    System.out.println("dic NO contiene HELLO como valor");
    }
    System.out.println("Mostrando todos los datos de la tabla hash...");
    Enumeration k = dic.keys();
    while( k.hasMoreElements() ) System.out.println( k.nextElement() );
    System.out.println("Mostrando todos los elementos de la tabla hash...");
```

```
        Enumeration e = dic.elements();
        while( e.hasMoreElements() ) System.out.println( e.nextElement() );
        System.out.println( "Eliminando el dato " + dic.remove("HOLA"));
        }
}
```


RESUMEN DEL CAPÍTULO

En este tema se estudia el concepto de vector o array. Existen numerosos problemas que se resuelven con estructuras de este tipo, por lo tanto es de obligado cumplimiento estudiarlo. También se estudia en profundidad las cadenas de caracteres que ya se vieron en capítulos anteriores. Estudiar arrays y no ver algoritmos de ordenación es un pecado, por lo tanto, se estudiarán dos de los más útiles como son el método de la burbuja, el cual es sencillo de comprender y el método de inserción directa. Existen multitud de algoritmos mucho más eficientes y rápidos que el de la burbuja. Se aconseja al alumno que busque alguno por Internet e intente comprenderlo.

EJERCICIOS RESUELTOS

1. Realiza un programa que genere una matriz 5 x 8 y muestre los elementos en forma de matriz.

```
public class matriz {
    public static void main(String[] args) {
    int[][] matriz = new int[5][8];
    for (int i=0;i<5;i++){
          for (int j=0;j<8;j++){
                matriz[i][j]=i+j;
          }
    }
    for (int i=0;i<5;i++){
          for (int j=0;j<8;j++){
                System.out.print(matriz[i][j]);
```

```
                System.out.print(" ");
            }
                System.out.println("");
        }
        }
}
```

2. Realiza un programa que cree un vector de 50 posiciones cargado con valores aleatorios. Los valores aleatorios deberán de estar entre el 1 y el 100. Una vez cargado el vector deberá de ordenarlo mediante el método de la burbuja y mostrarlo ordenado por pantalla.

Truco

Observa que para generar un número aleatorio se utiliza el método random de la clase Math el cual genera un número aleatorio (double) entre 0.0 y 1.0. Si este se múltiplica por LIMITE que es el rango de números a generar y se le suma 1, los números generados estarán siempre entre 1 y LIMITE.

```
public class burbuja {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
          return (int) (Math.random()*LIMITE+1);
    }
    public static void ordena(int array[]){
          int aux;
          for (int i=array.length;i>0;i--){
                for (int j=0;j<i-1;j++){
                      if (array[j]>array[j+1]){
                            aux = array[j+1];
                            array[j+1]=array[j];
                            array[j]=aux;
                      }
                }
          }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++){
                System.out.print(lista[i]+" ");
          }
    }
    public static void main(String[] args) {
```

```
            lista = new int[POS];
            for (int i=0;i<POS;i++){
                  lista[i]=getaleatorio();
            }
            muestra();//se muestra el vector desordenado
            System.out.println("");
            ordena(lista); //ordenación por burbuja
            System.out.println("");
            muestra();//se muestra el vector ordenado
            System.out.println("");
    }
}
```

3. Realiza un programa que cree un vector de 50 posiciones cargado con valores aleatorios. Los valores aleatorios deberán de estar entre el 1 y el 100. Una vez cargado el vector deberá de ordenarlo mediante el método del cocktail sort y mostrarlo ordenado por pantalla.

```
public class cocktail {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
          return (int) (Math.random()*LIMITE+1);
    }
    public static void ordenacocktail(int array[]){
      int i = 0, j = array.length - 1;
      while(i < j) {
         for(int k = i; k < j; k++) { //direccion ->
            if(array[k] > array[k + 1]) {
               int temp = array[k];
               array[k] = array[k + 1];
               array[k + 1] = temp;
            }
         }
         j--;
            for(int k = j; k > i; k--) { //direccion <-
               if(array[k] < array[k - 1]) {
                     int temp = array[k];
                     array[k] = array[k - 1];
                     array[k - 1] = temp;
               }
            }
         i++;
      }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++){
```

```
                System.out.print(lista[i]+" ");
            }
        }
        public static void main(String[] args) {
            lista = new int[POS];
            for (int i=0;i<POS;i++){
                lista[i]=getaleatorio();
            }
            muestra();
            System.out.println("");
            ordenacocktail(lista);
            System.out.println("");
            muestra();
            System.out.println("");
        }
}
```

4. Mejora el método ordenacocktail del ejercicio anterior y utiliza una variable swp como centinela o *flag*, de tal manera que ésta se active cuando hay algún intercambio. En el momento que no haya ningún intercambio el algoritmo debería de parar puesto que el vector ya estaría ordenado.

```
public static void ordenacocktail(int array[]){
      boolean swp = true;
      int i = 0, j = array.length - 1;
      while(i < j && swp) {
         swp = false;
         for(int k = i; k < j; k++) { //direccion ->
            if(array[k] > array[k + 1]) {
               int temp = array[k];
               array[k] = array[k + 1];
               array[k + 1] = temp;
               swp = true;
            }
         }
         j--;
         if(swp) {
            swp = false;
            for(int k = j; k > i; k--) { //direccion <-
               if(array[k] < array[k - 1]) {
                  int temp = array[k];
                  array[k] = array[k - 1];
                  array[k - 1] = temp;
                  swp = true;
               }
            }
         }
         i++;
```

```
        }
}
```

5. Tenemos una cadena notas con los nombres y las notas de 5 de los alumnos de clase. El contenido de la cadena es el siguiente:

"Juan Carlos\n 8.5\n Andrés\n 4.9\n Pedro\n 3.8\n Juan \n 6.3"

El formato es "nombre \n nota \n...."

Realiza un programa que muestre por pantalla por cada alumno lo siguiente:

El alumno X ha sacado la nota Y.

```
StringTokenizer notas;
notas = new StringTokenizer("Juan Carlos\n8.5\nAndrés\n4.9\n Pedro\n3.8\nJuan\
n6.3","\n");
while (notas.hasMoreTokens())
    System.out.println("El alumno "+notas.nextToken()+" ha sacado un "+notas.
nextToken());
```

6. Modifica el ejemplo del apartado 6.4.2 para que muestre los nombres ordenados.

```
public class test {
    private static String[]  lista;
    final static int POS=10; //número de posiciones del array
    public static void ordena(String array[]){
          String aux = new String();
          for (int i=array.length;i>0;i--){
                for (int j=0;j<i-1;j++){
                      if (array[j].compareTo(array[j+1])>0){
                            aux = array[j+1];
                            array[j+1]=array[j];
                            array[j]=aux;
                      }
                }
          }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++)    System.out.print(lista[i]+" ");
    }
    public static void main(String[] args) {
          lista = new String[POS];
          for (int i=0;i<POS;i++){
                String ln = System.console().readLine();
                lista[i]=ln.toString();
          }
```

```
            muestra(); //los muestra desordenados
            System.out.println("");
            ordena(lista); //ordena los nombres
            System.out.println("");
            muestra(); //ahora los muestra ordenados
            System.out.println("");
    }
}
```

7. Realiza un programa que muestre por pantalla los dos primeros argumentos tomados desde la línea de comandos.

Solución:

```
public class muestraArgs {
    public static void main(String[] args) {
            System.out.println("Primer argumento: " + args[0]);
            System.out.println("Segundo argumento: " + args[1]);
    }
}
```

8. Realiza un método *esCapicua* que tome como parámetro un entero y devuelva *true* si el número es capicua y *false* en caso contrario. Utiliza *wrappers* y objetos *String*.

Solución:

```
    public static boolean esCapicua(int dato){
            Integer i = new Integer(dato);
            String reverse = new StringBuffer(i.toString()).reverse().toString();
            return i.toString().equals(reverse.toString());
    }
```

9. Crea una clase *busquedabin* la cual tenga un método que busque un valor en un array ordenado utilizando la búsqueda binaria. Implementa el método de tal manera que muestre las posiciones (min y max) desde las cuales va buscando y los valores de dichas posiciones así como la posición intermedia y el valor de dicha posición.

```
public class busquedabin {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
            return (int) (Math.random()*LIMITE+1);
    }
```

```
    public static void ordena(int array[]){
        int aux;
        for (int i=array.length;i>0;i--){
            for (int j=0;j<i-1;j++){
                if (array[j]>array[j+1]){
                    aux = array[j+1];
                    array[j+1]=array[j];
                    array[j]=aux;
                }
            }
        }
    }
    public static void muestra(){
        for (int i=0;i<POS;i++){
            System.out.print(lista[i]+" ");
        }
    }
    public static int buscabin(int[] a, int valor, int min, int max){
        if (min == max) {
            System.out.println("SALIDA PORQUE MIN=MAX");
            return -1;
        }
        int mitad = (min + max)/2;
        System.out.println("min"+min+" a[min]"+a[min]+" max"+max+" a[max]"+a[max]+"
mitad"+mitad+" "+a[mitad]);
        if (valor == a[mitad]) return mitad;
        if (valor == a[min]) return min;
        if (valor == a[max]) return max;
        if (valor > a[mitad])
            return buscabin(a,valor,mitad+1,max);
        else
            return buscabin(a,valor,min,mitad-1);
    }
    public static void main(String[] args) {
        lista = new int[POS];
        for (int i=0;i<POS;i++){
            lista[i]=getaleatorio();
        }
        muestra();
        System.out.println("");
        ordena(lista);
        System.out.println("");
        muestra();
        System.out.println("");
        System.out.println(buscabin(lista,50,0,POS-1));
    }
}
```

El ejemplo anterior, como se puede observar genera y ordena la lista antes de hacer la búsqueda binaria.

- Realiza las siguientes modificaciones al ejercicio:
  - Modifica el ejercicio anterior para que no se puedan insertar valores repetidos en el array.
  - Modifica el ejercicio para que el algoritmo en vez de recursivo sea iterativo.

EJERCICIOS PROPUESTOS

- **1.** El ejercicio resuelto número 2, como podrás suponer, no es óptimo del todo. Imagínate que los números están ya ordenados (cosa muy improbable utilizando números aleatorios). Modifica el ejercicio anterior para que el procedimiento ordene y no dé más pasadas de las necesarias.

- **2.** Realiza un programa que cree dos vectores de 100 elementos. El primero almacenará una serie de datos numéricos desordenados. Dichos datos serán datos generados aleatoriamente. El segundo array contendrá los mismos datos pero ordenados por el método *insertion sort*.

- **3.** Modifica el código del ejercicio resuelto 5 para que las notas se almacenen en un vector de datos *double*.

- **4.** (Ejercicio de dificultad alta) Realiza un programa con el que puedas jugar a los barquitos con tu compañero. El programa simulará el juego clásico.

- **5.** (Ejercicio de dificultad alta) Programa que realice una sopa de letras. La sopa de letras tendrá un tamaño de matriz 15 x 15. El programa pedirá 10 palabras, las cuales las irá escondiendo de forma aleatoria por la matriz (obviamente las palabras siempre tendrán 15 letras o menos). Una vez escondidas las palabras rellenará las demás casillas de la matriz con letras de forma aleatoria. Solo se utilizarán mayúsculas. Si el usuario introduce palabras en minúsculas se transformarán a mayúsculas.

- **6.** Tenemos el siguiente método que indica cuándo un número es capicúa o no:

```
public static boolean esCapicua(int dato){
       Integer i = new Integer(dato);
       String reverse = new StringBuffer(i.toString()).reverse().toString();
       return i.toString()==reverse.toString();
}
```

Parece que no funciona porque todos los números que se introducen para comparar el método responde que no son capicúa. Descubre por qué no funciona y razona tu respuesta.

- **7.** Realiza un método que tome como parámetros de entrada dos arrays de enteros y devuelva como salida un único array con los elementos de los anteriores arrays ordenados de forma ascendente.

- **8.** Realiza un programa que cree 1000 números aleatorios y muestre los 10 mayores.

- **9.** Realiza un programa que cree un vector de 100 posiciones con números aleatorios entre 10 y 80. Una vez creado el vector el programa deberá mostrar el mayor, el menor, el valor que más se repite y la media.

- **10.** Realiza un programa que cree un vector de 100 posiciones con números aleatorios entre 1 y 100. Una vez creado el vector el programa deberá ordenar el vector y mostrar los números entre 1 y 100 que no han sido almacenados.

8 Bases de datos relacionales

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender la arquitectura JDBC.
- ✓ Crear aplicaciones que almacenen o recuperen datos de una base de datos.
- ✓ Manejar excepciones creadas por las llamadas a métodos JDBC (SQLExceptions).
- ✓ Realizar operaciones básicas sobre las bases de datos (insertar, modificar y borrar datos).
- ✓ Realizar operaciones avanzadas sobre bases de datos como llamadas a procedimientos almacenados, ejecución de transacciones, etc.

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 realizar consultas SQL, recuperar datos de la base de datos y realizar cambios a la base de datos a través del *datasource* u 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 (testear aplicaciones web o para comunicarse con bases de datos vía ODBC).

8.1 LA ARQUITECTURA JDBC

Existen dos modelos fundamentales en sistemas informáticos que acceden a bases de datos:

- Modelo de arquitectura basada en **dos niveles (two-tier).**

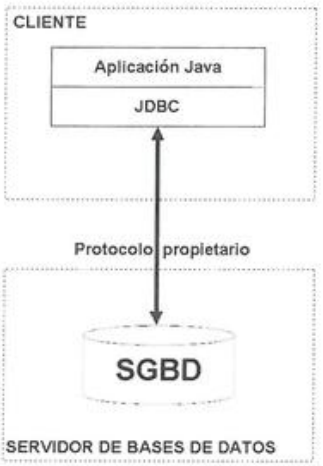


Figura 8.1. Arquitectura JDBC two-tier

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).**

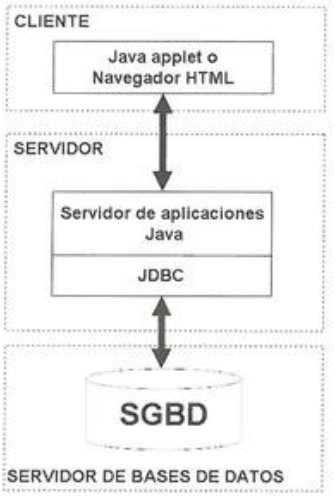


Figura 8.2. Arquitectura JDBC three-tier

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 ganando en muchas ocasiones una mejora del rendimiento. Cada vez mas 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.

8.1.1 QUÉ SE NECESITA PARA TRABAJAR CON BASES DE DATOS Y JDBC

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 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 esta 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 y de esa manera instalo Apache, MySQL, PHP y PhpMyAdmin. Eso me permite administrar y manejar la base de datos desde un navegador web.

8.2 CONEXIONES CON BASES 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:

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: "andrés".

  La *password* del anterior usuario: "pelusilla".

8.3 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 como 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.

```
C:\WINDOWS\system32\cmd.exe
' in 'field list'
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Me
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown So
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknow
rce)
        at java.lang.reflect.Constructor.newInstance(Unknown Source)
        at com.mysql.jdbc.Util.handleNewInstance(Util.java:406)
        at com.mysql.jdbc.Util.getInstance(Util.java:381)
        at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1030)
        at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3558)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3490)
        at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1959)
        at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2109)
        at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2637)
        at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2566)
        at com.mysql.jdbc.StatementImpl.executeQuery(StatementImpl.java:146
        at test.viewTable(test.java:26)
        at test.main(test.java:48)
SQLState: 42S22
Error Code: 1054
```

Figura 8.3. Tratamiento de una excepción Java

La figura anterior muestra el resultado del tratamiento de una excepción Java.

8.4 CREACIÓN Y CARGA DE DATOS EN TABLAS

Para los siguientes ejemplos se va a crear la siguiente estructura de tablas.

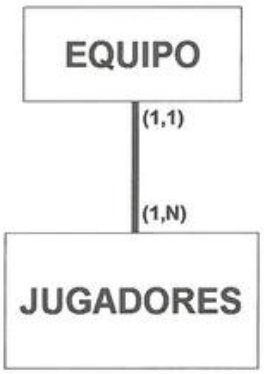


Figura 8.4. Relación equipo-jugadores

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 una serie de 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));
```

8.4.1 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.

A FONDO

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 como 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 pueden 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*, *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.

8.4.2 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();
      }
}
```

8.5 RECUPERACIÓN DE INFORMACIÓN

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* el cual 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*.

A FONDO

EL INTERFACE 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 detrás (*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 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* puede ser actualizado utilizando el interface *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.

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* **SI** 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.

8.5.1 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();
    }
}
//** Fin código****
```

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*.

A FONDO

LOS CURSORES

Los cursores en JDBC son los anteriormente vistos *ResulSet*. Cuando se crea un *ResulSet* este se posiciona 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 esta 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 esta 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*.

8.6 MODIFICACIÓN Y ACTUALIZACIÓN DE LA BASE DE DATOS

8.6.1 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();
      }
  }
```

8.6.2 MODIFICAR 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 y 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.

8.6.3 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*. En el siguiente método se puede observar como se pueden realizar inserciones de datos en una base de datos. Al ejecutar el método *inserRow()* del objeto *ResultSet* se insertarán los datos tanto en el *ResultSet* como en la base de datos.

8.7 OTRAS OPERACIONES SOBRE BASES DE DATOS RELACIONALES

En este apartado se van a ver otro tipo de operaciones con bases de datos como pueden ser las transacciones, necesarias muchas veces cuando queremos realizar un grupo de comandos SQL en bloque y funciones y procedimientos almacenados las cuales son sumamente útiles a la hora de simplificar código en los programas y para estandarizar ciertas operaciones.

8.7.1 TRANSACCIONES

Existen ocasiones en las que se necesita que las operaciones se ejecuten en bloque, es decir, que necesitamos que se ejecuten o todas las operaciones o ninguna porque si no la base de datos se quedará en un estado inconsistente.

Figura 8.5. Transacción entre cuentas bancarias

Imaginemos una transferencia de dinero entre dos cuentas bancarias. La primera operación es restar de la cuenta A la cantidad a transferir para luego ingresarla en la cuenta B. En el caso de que haya algún problema para actualizar el saldo de la cuenta B y esta operación no se realice, nos podemos encontrar con que en la cuenta A se ha retirado una cantidad de dinero y en la cuenta B no se ha hecho efectiva la transferencia. La base de datos en ese momento se quedaría inconsistente. En ese caso se necesita realizar una transacción que englobe la retirada de efectivo de la cuenta A y el ingreso de ese dinero en la cuenta B. Esas operaciones se realizarán en bloque y en caso de encontrar algún problema no se realizará ninguna.

En todos los ejemplos que hemos visto hasta ahora, cuando se realiza una operación de modificación de base de datos (borrado, inserción o actualización) los cambios se producen cuando la sentencia se ejecuta. No hace falta ejecutar una orden para actualizar cambios en la base de datos. Esto es así porque está habilitado el modo **auto-commit** en la conexión con la base de datos.

Para deshabilitar el modo *auto-commit* en la base de datos hay que ejecutar la siguiente sentencia:

```
con.setAutoCommit(false);
```

Donde con es nuestro objeto conexión.

Vamos a ver un ejemplo práctico de cómo funcionaría todo esto:

numcuenta	saldo
1	1000
2	200

Figura 8.6. Cuentas bancarias y saldo

Tenemos en la tabla cuentas dos cuentas con los saldos que aparecen en la figura anterior. El propietario quiere realizar una transferencia de dinero entre ambas cuentas de 500 euros (de la cuenta 1 a la cuenta 2).

En el siguiente método se muestra como se realizaría una transacción en una base de datos de una cuenta a otra.

```
public static void transaccion(Connection con, String BDNombre,int cuentaA,int cuentaB,int
cantidad) throws SQLException {
    Statement stmt = null;
    String actualizaA = "update " + BDNombre + ".CUENTAS " +
                                "set SALDO = SALDO - "+cantidad+" where NUMCUENTA =
"+cuentaA;
    String actualizaB = "update " + BDNombre + ".CUENTAS " +
                                "set SALDO = SALDO + "+cantidad+" where NUMCUENTA =
"+cuentaB;
    try {
      con.setAutoCommit(false);
      stmt = con.createStatement();
      stmt.executeUpdate(actualizaA);
      stmt.executeUpdate(actualizaB);
    } catch (SQLException e ) {
      printSQLException(e);
      if (con != null) {
        try {
          System.err.print("Roll back de la Transaccion");
          con.rollback();
        } catch(SQLException excep) {
          printSQLException(excep);
        }
      }
    } finally {
      stmt.close();
      con.setAutoCommit(true);
    }
 }
//** Fin código****
```

Nótese como se cambia el estado *auto-commit* de la base de datos primero a *false* y luego a *true* (el valor por defecto) y como si ocurre algún error en la transacción se invoca al método *rollback()* del objeto conexión.

La llamada al método para la realización de la transferencia de 500 euros desde el método principal sería la siguiente:

```
//** Inicio código****
public static void main(String[] args) {
    Connection con;
    try {
con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test","root","");
    transaccion(con,"test",1,2,500);
    }
    catch (Exception e) {
    e.printStackTrace();
    }
}
```

A FONDO

LAS TRANSACCIONES

Imaginemos que, como en el ejemplo anterior, estamos transfiriendo una cantidad de 500 euros de la cuenta A a la cuenta B mientras que en la cuenta A se está haciendo un ingreso por valor de 200 euros. El sistema retirará 500 euros de la cuenta A y antes de terminar la transacción ingresa 200 euros en esa misma cuenta. En este momento la cuenta A dispone de 700 euros (100 – 500 + 200). Por la razón que sea, el ingreso de 500 euros a la cuenta B se aborta y el sistema tiene que abortar la transacción. En este caso, el sistema dejará el estado de la cuenta A tal y como estaba antes de comenzar la transacción, con un saldo de 100 euros.

En este caso tenemos un fallo del sistema. 200 euros que fueron ingresados a la cuenta A se han perdido con los consiguientes problemas que nos puede acarrear este mal funcionamiento de nuestro sistema. A estas lecturas de datos obsoletos se le suele llamar **dirty reads** (lecturas sucias).

Para prevenir este tipo de problemas, los gestores de bases de datos prevén un sistema de bloqueos que hacen que las filas implicadas en las transacciones no puedan ser modificadas. Cuando trabajábamos en modo **auto-commit** el bloqueo era a nivel de una sola fila y no había problema, ahora las transacciones pueden comprometer a muchas filas y a varias tablas.

Existen fundamentalmente 3 problemas o inconsistencias de datos cuando se trabaja con transacciones con bases de datos:

Lecturas sucias. Es cuando se realiza una lectura que no tiene un valor permanente. La transacción A modifica el valor *x* de una tabla. Mientras la transacción se está ejecutando B lee el valor *x'* de la tabla ya modificado. Si por la razón que sea la transacción A se aborta, el valor *x'* leído habrá sido un valor ficticio y no tiene ninguna validez.

Lecturas no repetibles. Se producen cuando se hacen varias lecturas de datos en una transacción y los datos ya no son los mismos sin haber sido modificados por la transacción. Por ejemplo, la transacción A lee un dato *x* de una tabla. Después de eso la transacción B modifica dicho valor. La transacción A vuelve a leer el dato *x'* de la tabla pero ya no es el mismo. El dato ha cambiado de valor y el resultado de la transacción A es imprevisible.

Lecturas fantasma. Se produce cuando una transacción A recupera una serie de filas de una tabla o tablas que cumplen una condición. Si la transacción B realiza una modificación de alguna de las tablas implicadas (*update*, *delete* o *insert*), es posible que cuando la transacción A realice la consulta obtenga nuevas filas o pierda algunas de las filas que estaban antes (aparecen o desaparecen datos).

Para evitar este tipo de problemas, JDBC utiliza el nivel de aislamiento (existen 5 niveles de aislamiento) al realizar las transacciones. En la siguiente tabla se puede ver el nivel de aislamiento y los problemas que resuelve cada uno de ellos:

Nivel de Aislamiento	Transacciones	Lecturas sucias	Lecturas no repetibles	Lecturas fantasma
TRANSACTION_NONE	NO SOPORTADO	NO APLICABLE	NO APLICABLE	NO APLICABLE
TRANSACTION_READ_COMMITED	SOPORTADO	NO	SI	SI
TRANSACTION_READ_UNCOMMITED	SOPORTADO	SI	SI	SI
TRANSACTION_REPETEABLE_READ	SOPORTADO	NO	NO	SI
TRANSACTION_SERIALIZABLE	SOPORTADO	NO	NO	NO

Figura 8.7. Nivel de aislamiento de las transacciones

Consejo

El método *rollback* generalmente se utiliza cuando se lanza una excepción. En el momento que se lanza la excepción esto nos indica que algo ha pasado pero nunca nos va a decir qué datos se han modificado y cuáles no. Por lo tanto, en muchas ocasiones el rollback sirve para "curarse en salud" y hacer "borrón y cuenta nueva".

8.7.2 FUNCIONES DE USUARIO

En SQL es muy común utilizar funciones en los comandos como por ejemplo count(*), sum(columna), avg(columna), max(columna), etc. En ocasiones, el programador necesita realizar operaciones muy concretas que devuelven un valor y en ese caso se utilizan funciones almacenadas. Estas funciones almacenadas son muy parecidas a los procedimientos almacenados aunque su forma de utilización es diferente. Las diferencias entre funciones y procedimientos almacenados son las siguientes:

- Las funciones siempre devuelven un valor mientras que los procedimientos no. El tipo de este valor se define cuando se declara la función. Obviamente este tipo de dato tiene que ser un tipo de datos de la base de datos (en nuestro caso MySQL).
- Desde una sentencia SQL se puede llamar a una función pero no a un procedimiento almacenado.
- Las funciones devuelven un valor pero nunca un *Resultset*.
- Las funciones tienen solo parámetros de entrada. El único parámetro de salida por así decirlo es el dato que devuelve la función (aunque el valor de salida no se puede considerar como parámetro).

Recuerda

La sintaxis de las funciones y procedimientos almacenados difieren de un SGBDR a otro, incluso de una versión a otra podrían cambiar. Consulta los manuales de referencia de cada gestor cuando vayas a realizar alguna función o procedimiento almacenado.

La sintaxis de las funciones almacenadas en MySQL es la siguiente:

```
CREATE FUNCTION nombre_de_función (parámetro1, parámetro2, ..., parámetroN )
    RETURNS tipo ( CHAR|INT|FLOAT|DECIMAL|… )
    BEGIN
        .......
    RETURN valor
END
```

Un ejemplo de función almacenada en MySQL es el que se define a continuación. En esta función almacenada se recuenta y se devuelve el número de filas que existe en la tabla jugadores.

```
DELIMITER //
CREATE FUNCTION dimeCuantos()
RETURNS INT
BEGIN
DECLARE j INT;
SELECT   COUNT(*) INTO   j   FROM   jugadores;
RETURN j;
END //
```

Nótese que se utiliza como delimitador "//" dado que las sentencias SQL utilizan el delimitador por defecto ';'.

Para comprobar si funciona nuestra función se puede ejecutar la siguiente consulta SQL:

```
SELECT dimeCuantos() FROM DUAL;
```

La tabla dual es una tabla que solo tiene una fila y, por lo tanto, el resultado aparecerá en solo una fila. Si hacemos esa sentencia pero para la tabla jugadores, nos saldrá tantos resultados como filas tenga la tabla jugadores. Prueba a ejecutar la anterior sentencia y la siguiente con más de un jugador en la tabla jugadores y verás la diferencia:

```
SELECT dimeCuantos() FROM JUGADORES;
```

El siguiente método muestra la forma de utilización de funciones almacenadas en una base de datos:

```
public static void getJugadores(Connection con) throws SQLException {
    Statement stmt = null;
    String query = "select dimeCuantos() FROM DUAL";
    try {
      stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String cuantos = rs.getString(1);
        System.out.println("Existen  "+cuantos+" jugadores en nuestra base de datos");
        System.out.println("************************************************");
      }
```

```
        } catch (SQLException e ) {
              printSQLException(e);
        } finally {
          stmt.close();
        }
    }
```

8.7.3 PROCEDIMIENTOS ALMACENADOS

Un procedimiento almacenado es una serie de comandos que realizan una tarea concreta en una base de datos. Los procedimientos almacenados generalmente se crean cuando existe una tarea que se va a realizar muchas veces.

En ocasiones es bueno utilizar procedimientos almacenados para estandarizar tareas complejas en una base de datos. Imaginemos que en un banco tenemos una serie de clientes a los cuales se les va a dar una gratificación, descuento o prima dependiendo de los productos que tengan contratados, saldo medio... Si esa operación se va a realizar desde varias aplicaciones es posible que alguna de ellas lo calcule de forma diferente. Si se estandariza y se incluye en un procedimiento almacenado el proceso se centraliza y estaremos seguros que no existen diferentes formas de hacer lo mismo.

La sintaxis a grandes rasgos es la siguiente:

```
CREATE PROCEDURE nombre_del_procedimiento ([PARAM1[,...]])
BEGIN
CÓDIGO_DEL_PROCEDIMIENTO
END
```

Los parámetros se definen de la siguiente manera:

```
[ IN | OUT | INOUT ] NOMBRE  TIPO_DE_DATO
```

En el siguiente código se muestra un procedimiento almacenado. Se ha creado un procedimiento similar a la función almacenada del apartado anterior.

```
DELIMITER //
CREATE PROCEDURE cuantosJugadores(OUT c INT)
     BEGIN
       SELECT COUNT(*) INTO c FROM jugadores;
END //
```

En este caso el procedimiento almacenado tiene un parámetro de salida (OUT) y devuelve el número de filas de la tabla jugadores. El programa Java que ejecute dicho procedimiento deberá de recoger dicho valor de salida.

```
public static void getJugadoresProc(Connection con) throws SQLException {
    try {
```

```
        CallableStatement cs = con.prepareCall("{call cuantosJugadores (?)}");
        cs.registerOutParameter(1,Types.INTEGER);
        cs.execute();
        int cuantos = cs.getInt(1);
        System.out.println("Existen  "+cuantos+" jugadores en nuestra base de datos");
      System.out.println("*************************************************");
    } catch (SQLException e ) {
          printSQLException(e);
    }
}
```

En el ejemplo anterior se puede observar cómo se invoca el procedimiento almacenado cuantosJugadores desde el método *getJugadoresProc*.

Anteriormente hablamos de parámetros de entrada, salida y entrada/salida. La llamada a procedimientos almacenados con diferentes tipos de parámetros no es la misma. En el siguiente ejemplo se puede ver cómo se realizarían dichas llamadas:

```
try {
    CallableStatement cs;
  // Llamada a un procedimiento sin parámetros
    cs = con.prepareCall("{call miproc}");
    cs.execute();
  // Llamada a un procedimiento con un parámetro OUT
    cs = con.prepareCall("{call miproc_out(?)}");
    // Hay que registrar el parámetro OUT con su tipo
    cs.registerOutParameter(1, Types.VARCHAR);
    // Ejecutar el procedimiento y recupearar el parámetro
    cs.execute();
    String dato = cs.getString(1);
  // Llamada a un procedimiento con un parámetro IN
    cs = con.prepareCall("{call miproc_in(?)}");
    // Hay que actualizar el valor del parámetro IN
    cs.setString(1, "HOLA");
    // Execute the stored procedure
    cs.execute();
  // Llamada a un procedimiento con un parámetro IN /OUT
    cs = con.prepareCall("{call miproc_inout(?)}");
    // Hay que registrar el parámetro IN/OUT con su tipo
    cs.registerOutParameter(1, Types.VARCHAR);
    // Hay que actualizar el valor del parámetro IN/OUT
    cs.setString(1, "HOLA");
    // Execute the stored procedure and retrieve the IN/OUT value
    cs.execute();
```

```
        dato = cs.getString(1);          // OUT parameter
    } catch (SQLException e) {
            printSQLException(e);
    }
```


La clase CallableStatement
Los objetos de la clase *CallableStatement* son utilizados en Java para llamar a procedimientos almacenados de la base de datos.

A FONDO

JNI (JAVA NATIVE INTERFACE)

JNI es un mecanismo que permite al programador que sus programas escritos en Java ejecuten código nativo (programas escritos en ensamblador, C o C++) y viceversa.

Existen ocasiones en las que las bibliotecas estándar de Java no soportan ciertas características (control de sonido, lectura/escritura de ficheros, etc.) y es necesario acudir a aplicaciones externas a Java, como programas en C o C++. Las aplicaciones entonces al utilizar JNI se vuelven dependientes de la plataforma, por lo tanto no tiene sentido utilizar JNI si determinadas funcionalidades son proporcionadas por Java dado que son seguras e independientes de la plataforma.

Los programas en C o C++, al ser compilados para la máquina, son mucho más rápidos y eficientes que los programas en Java, por lo tanto, cuando se necesita velocidad de ejecución se acude a JNI para ejecutar programas nativos más rápidos.

Por lo general, se suele utilizar JNI para llamar a funciones de librerías nativas aunque también podría hacerse lo contrario, por ejemplo incrustar una VM de Java en un navegador Web escrito en C o C++ y realizar llamadas a métodos que se ejecuten en la máquina virtual.

Como se ha podido ver, JNI es un API compleja y desarrollar aplicaciones que utilicen JNI es una tarea de programadores experimentados por lo que no tiene sentido incluirlo en el temario del libro.

La utilización de JNI con C++ es mucho más sencilla que utilizarlo en C, dado que C++, al igual que Java, son lenguajes de Programación Orientados a Objetos.

RESUMEN DEL CAPÍTULO

Las bases de datos son un elemento de almacenamiento de información básico en cualquier aplicación. Es posible ver también aplicaciones que utilicen ficheros para almacenar datos, pero las bases de datos generalmente son más eficientes y requieren de un menor esfuerzo al programar, dado que nos brindan todo tipo de posibilidades para filtrar y tratar la información. Existen varias formas de acceder a las bases de datos pero en este tema se estudia la arquitectura JDBC, la cual es muy utilizada por su facilidad de uso y su practicidad. Al final del capítulo se verán opciones más avanzadas en el manejo de bases de datos desde Java, como son las transacciones y el uso de funciones y procedimientos de usuario.

EJERCICIOS RESUELTOS

1. Necesitamos incorporar una serie de datos procedentes de un fichero a una base de datos. El fichero tiene el siguiente formato:

Formato

matrícula marca modelo color año precio

Datos ejemplo

8012-CLY RENAULT MEGANE NEGRO 2003 2350

5068-GDB VOLKSWAGEN PASSAT GRIS 2008 13500

3268-BVN OPEL ASTRA NEGRO 2002 2000

La base de datos se llama concesionario y la tabla se llama coches pero no está creada. Se pide:

- Programar un método *creaTablacoches()* el cual cree la estructura de la tabla coches con el siguiente formato:

Tabla 8.1. Estructura de la tabla coches

Coches	
Matrícula (PK)	varchar(8) NOT NULL
Marca	(40) NOT NULL
Modelo	varchar(40) NOT NULL
Color	varchar(40) NOT NULL
Anio	integer NOT NULL
Precio	integer NOT NULL

Solución:

```
public static void creaTablacoches(Connection con, String BDNombre) throws SQLException
{
    String createString = "create table " + BDNombre + ".COCHES " +
                  "(MATRICULA char(8) NOT NULL," +
                  "MARCA varchar(40) NOT NULL," +
                  "MODELO varchar(40) NOT NULL," +
                  "COLOR varchar(20) NOT NULL," +
                  "ANIO integer NOT NULL," +
                  "PRECIO integer NOT NULL," +
                  "PRIMARY KEY (MATRICULA))";
    Statement stmt = null;
    try {
      stmt = con.createStatement();
      stmt.executeUpdate(createString);
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      stmt.close();
    }
}
```

- Programar un método *cargaTablacoches()* el cual cargue la tabla coches con los datos contenidos en un fichero.

Solución:

```
public static void cargaTablacoches(Connection con, String BDNombre, String archivo)
throws SQLException {
    Statement stmt = null;
    try {
      stmt = con.createStatement();
           File fe = new File(archivo);
           FileReader fr = new FileReader(fe);
           BufferedReader br = new BufferedReader(fr);
           String s;
           while((s = br.readLine()) != null) {
                  StringTokenizer str;
                  str = new StringTokenizer(s);
                  String comando ="INSERT INTO " + BDNombre + ".COCHES VALUES ("
                  +"`"+str.nextToken()+"`," //MATRICULA
                  +"`"+str.nextToken()+"`," //MARCA
                  +"`"+str.nextToken()+"`," //MODELO
                  +"`"+str.nextToken()+"`," //COLOR
                  +""+str.nextToken()+"," //AÑO
                  +""+str.nextToken()+"" //PRECIO
                  +")";
                  stmt.executeUpdate(comando);
           }
           if (fr != null) fr.close();

    }catch (FileNotFoundException fnf) {
                  System.err.println("Fichero no encontrado " + archivo);
    }catch(IOException e){
                  System.err.println("Se ha producido una IOException");
                  e.printStackTrace();
    }catch (SQLException e) {
                  e.printStackTrace();
    }catch (Throwable e) {
                  System.err.println("Error de programa: " + e);
                  e.printStackTrace();
    }finally {
                  stmt.close();
    }
}
```

En el siguiente código se muestra un método *main()* con la llamada a estos métodos:

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

```
    con = DriverManager.getConnection("jdbc:mysql://localhost:3306/concesionario","admi
ndb","carratraca");
    creaTablacoches(con," concesionario ");
    cargaTablacoches(con," concesionario ","coches.txt");
    }
    catch (Exception e) {
    e.printStackTrace();
    }
}
```

- Modifica el ejercicio anterior para que el método *cargaTablacoches()* evite cargar vehículos cuyas matrículas ya existen en la tabla coches. Utiliza un nuevo método *existeCoche()* que devuelva un valor *booleano* indicando si la matrícula existe o no en la base de datos.

EJERCICIOS PROPUESTOS

- **1.** Con los datos del primer ejercicio propuesto, se pide crear una interfaz gráfica para poder visualizar los datos de los coches en stock en el concesionario. La interfaz deberá de tener el siguiente aspecto:

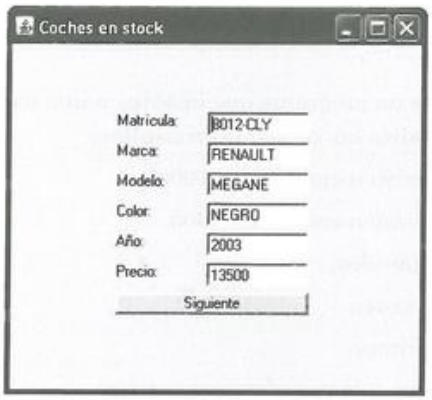


Figura 8.8. Aspecto de la aplicación de coches

No olvides deshabilitar el botón una vez se haya mostrado el último coche disponible.

- **2.** El concesionario nos ha pedido aumentar la funcionalidad de la interfaz anteriormente programado. Ahora nos piden crear los botones *Insertar* y *Borrar*. El usuario podrá borrar los datos del coche que se está visualizando o insertar nuevos datos en los campos correspondientes y añadirlos a la base de datos. El botón insertar previamente deberá comprobar antes de insertar el coche que éste no exista en la base de datos. El aspecto de la interfaz será parecido al siguiente:

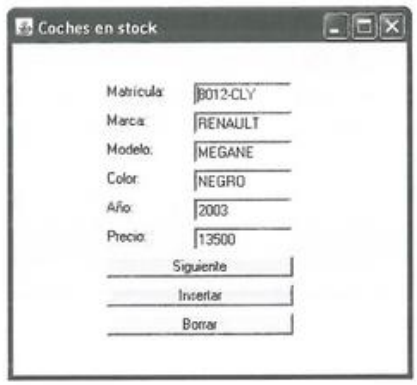


Figura 8.9. Aspecto de la aplicación mejorada de coches

- **3.** Añádele más funcionalidad al ejercicio anterior. Puedes intentar añadir el botón *Limpiar Campos* o incluir la posibilidad de poder cargar ficheros de datos desde la interfaz.

- **4.** Realiza un programa con una interfaz gráfica que permita insertar en una tabla de una base de datos los resultados de los sorteos de la lotería primitiva. Recuerda que tendrá que almacenar la jornada, la fecha y los cinco números más el complementario. El programa deberá comprobar que todos los campos están cumplimentados y que ningún número se repite.

- **5.** Realiza un programa con una interfaz gráfica que permita comprobar un boleto de primitiva de una jornada determinada. El programa deberá comprobar los números con los almacenados en la base de datos creada en el ejercicio anterior.

- **6.** (Ejercicio de dificultad alta) Realiza un programa que inserte en una base de datos los números primos del 1 al 1000. Una vez insertados los datos realiza las siguientes consultas:
  - Mostrar cuántos números primos existen entre 500 y 1000.
  - Mostrar cuántos números primos existen entre 100 y 900.
  - Mostrar cuál es el número primo más alto.
  - Mostrar cuántos números primos existen.
  - Mostrar la media de los números primos.
  - Mostrar cuántos números primos son a la vez números vampiro.
- **7.** Necesitamos crear una pequeña aplicación que gestione los animales de un pequeño zoológico. Cada animal tendrá las siguientes características:
  - Nombre.
  - Edad.
  - Color.
  - TipoComida.
  - CantidadComida (cantidad de comida en Kgs que toma diariamente).

- Cuidador.
- Características especiales (en este campo se anotarán características del propio animal como por ejemplo si está enfermo, si está a dieta de algún tipo, etc.).
- Ubicación.
- Número interno (codigo identificativos del animal, parecido al DNI).

El programa deberá poder Insertar, Modificar y Eliminar animales en la base de datos. Cuando se inserta un animal el sistema deberá de dotarle de un código interno.

Se deberán poder realizar consultas como:

- Listado de animales de una ubicación determinada.
- Cuántos animales hay por ubicación.
- Tipo de comida y cantidad de comida que comen los animales.
- Listado de animales responsabilidad de un cuidador determinado.
- Algunos cuidadores tienen el título de veterinario. Se necesitaría también saber el número de veterinarios por ubicación.

■ 8. Realiza una pequeña interfaz gráfica para el ejercicio anterior. El diseño lo hará el alumno dependiendo de las características del programa. Intenta realizar un diseño de interfaz que no solo cumpla las características pedidas, sino que pueda ser ampliable en un futuro con nuevas funcionalidades.

9 Persistencia de los objetos en bases de datos orientadas a objetos

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender el concepto de base de datos orientada a objetos.
- ✓ Reconocer las diferencias entre un lenguaje SQL y los sistemas de recuperación de datos de una BDOO.
- ✓ Conocer las diferencias existentes entre una BD relacional y una BD Orientada a Objetos.
- ✓ Trabajar con una verdadera base de datos OO.
- ✓ Realizar pequeñas aplicaciones con Java y bases de datos OO.
- ✓ Conocer las diferencias entre los distintos sistemas de recuperación de datos en db4o (SODA, NQ y QBE).

Nota importante
Para todos los ejemplos de este tema hemos utilizado la última versión disponible de db4o que es la 8.0 beta. Es posible que debido al cambio de versiones haya que modificar ligeramente el código para adecuarse a las nuevas actualizaciones. Siempre que utilices una versión de esta base de datos no olvides trabajar con la documentación de la versión. Para realizar prácticas y ejecutar los ejemplos, alumno encontrará la versión 8.0 en el material adicional suministrado junto con el libro.

9.1 BASES DE DATOS ORIENTADAS A OBJETOS

Las bases de datos orientadas a objetos (BDOO) están, como su nombre indica especialmente diseñadas para trabajar con datos de tipo *objeto* mientras que las bases de datos tradicionales la filosofía es totalmente distinta. Estas bases de datos tradicionales (generalmente relacionales) siguen los modelos clásicos de datos mientras que en los modelos Orientados a Objetos los datos manejados por la base de datos serán clases y objetos.

Uno de los problemas que se encuentran los programadores en los lenguajes orientados a objetos es la necesidad de almacenar y recuperar los datos de una forma eficiente y sencilla. Si se utilizan modelos relacionales se pierde parte de las ventajas de trabajar con Orientación a Objetos puesto que hay que hacer una transformación entre objetos y datos relacionales (y viceversa). Esta diferencia de paradigmas fue la que impulso a crear los sistemas gestores de bases de datos orientados a objetos (SGBDOO) para evitar las dificultades del paso del modelo de objetos al modelo relacional, se buscaba la eficiencia y la sencillez. ¿Por qué perder tiempo en rehacer las estructuras de datos (objetos) cuando se pueden almacenar y recuperar directamente?

9.1.1 BASES DE DATOS ORIENTADAS A OBJETOS COMERCIALES

A continuación se enumeran algunas Bases de Datos Orientadas a Objetos disponibles en el mercado:

- **Objetivity/DB**. Es una Base de Datos Orientada a Objetos que ofrece soporte para Java, C++, Python y otros lenguajes. Ofrece un alto rendimiento y escalabilidad. No es un producto libre aunque ofrecen versiones de prueba durante un periodo determinado.
- **db4o**. Es una Base de Datos Open Source para Java y .NET. Se distribuye bajo licencia GPL.
- **Intersystems Cache®**. Es una Base de Datos Orientada a Objetos que ofrece soporte para Java, C++, .NET, etc. Se vanaglorian de poder manejar grandes volúmenes de información y ejecutar sentencias SQL de forma más rápida que algunas bases de datos relacionales. Todo esto con un consumo mínimo de recursos y de hardware.
- **EyeDB**. Sistema gestor de Bases de Datos Orientado a Objetos (OODBMS) basado en la especificación ODMG el cual está desarrollado y soportado por la compañía francesa SYSRA. Proporciona un modelo avanzado de objetos (herencia, colecciones, arrays, métodos, *triggers*, *constraints*, etc.), un lenguaje de definición de objetos basado en ODMG ODL, un lenguaje de manipulación y consulta de datos basado en ODMG OQL e interfaces de programación para C++ y Java. Se distribuye bajo la licencia GNU (*GNU lesser General Public License*). Es un software libre.

9.2 CARACTERÍSTICAS DE LAS BASES DE DATOS ORIENTADAS A OBJETOS

Las **características** de las Bases de Datos Orientadas a Objetos son análogas a los lenguajes de programación OO. Algunas de las características relevantes de una Base de Datos Orientada a Objetos son las siguientes:

- Se entienden las Bases de Datos OO como un sistema de modelado del mundo real. Cada entidad del mundo real será un objeto en la base de datos.
- Los objetos tienen un identificador único que los identifica y los diferencia de los demás objetos del mismo tipo. Eso implica por ejemplo, que cada vez que se quiera modificar un objeto se tenga que recuperar de la base de datos, modificar y almacenar nuevamente. Esta operación es totalmente diferente en los SGBDR.
- Pueden existir objetos con la misma información pero con diferente identidad. Es más, cuando se modifican los valores de los atributos, el objeto sigue siendo el mismo.
- Es posible almacenar objetos complejos en la base de datos sin que por ello haya que realizar operaciones especiales sobre la base de datos.
- El concepto de herencia se mantiene incluso en la base de datos.
- Es el usuario el que modela los objetos de la base de datos y etiqueta los atributos y métodos que son visibles en la interfaz del objeto y cuáles no.
- El SGBDOO se encarga de acceder a los miembros de los objetos sin necesidad de escribir métodos para acceder a ellos.
- Acceso rápido a los datos dado que no hace falta realizar *joins* de tablas.
- Control de versiones. Algunos SGBDOO permiten un control de versiones.
- Implantación de conceptos del modelo OO como (polimorfismo, sobrecarga, sobrescritura, etc.).

Por lo tanto, dadas las características anteriores las Bases de Datos Orientadas a Objetos se hacen más apropiadas cuando tenemos una gran cantidad de tipos de datos diferentes, objetos con comportamientos avanzados o con un gran número de relaciones entre ellos.

Entre las **ventajas** que obtenemos al trabajar con las Bases de Datos Orientadas a Objetos se encuentran las siguientes:

- La primera y fundamental es el no tener que reensamblar los objetos cada vez que se accede a la base de datos. El resultado de las consultas son objetos por lo tanto la velocidad de procesamiento se aumenta.
- Cuando cambia un objeto la forma de actualizarlo en la base de datos es simplemente almacenándolo. Esta acción suele ser una secuencia simple de comandos en el código.
- La reutilización que es una de las características de los lenguajes de programación orientados a objetos se mantiene con lo que se mejoran los costes de desarrollo.
- El acceso a la información a través de objetos en una aplicación desarrollada como tal es más natural que hacerlo a través de tablas y filas.
- El control de acceso y concurrencia se facilita enormemente dado que se puede bloquear el acceso a ciertos objetos incluso en una jerarquía completa de objetos.

- Estos sistemas funcionan de forma eficiente en entornos cliente/servidor y arquitecturas distribuidas.
- No hace falta redefinir las relaciones entre objetos pues ya existen en el modelo de objetos y la base de datos se encarga de hacer persistente lo ya diseñado.

Por otra parte las limitaciones de las Bases de Datos Orientadas a Objetos serán las siguientes:

- En las bases de datos existe el lenguaje SQL que es un estándar bastante asentado y fuerte. En las bases de datos orientadas a objetos esto no ocurre.
- La estructura de las bases de datos relacionales es simple y fácil de entender al contrario que la de las bases de datos orientadas a objetos.
- Existen aplicaciones que aunque están escritas con lenguajes orientados a objetos, en ciertas ocasiones es más eficiente almacenar los datos en bases de datos relacionales debido a las consultas que se van a realizar sobre ellas.
- Se reduce la velocidad de acceso debido a que la Base de Datos Orientada a Objetos tiene que tener en cuenta las relaciones de herencia entre clases.
- En ocasiones aunque se gana en simplicidad en el manejo de la base de datos desde los lenguajes de programación orientados a objetos, cuando se trata de realizar consultas complejas resulta más adecuado una base de datos relacional accedida mediante SQL.

9.3 INSTALACIÓN DEL GESTOR DE BASES DE DATOS

Para trabajar con una Base de Datos Orientada a Objetos se ha elegido **db4o**, la cual es una base de datos con licencia **GPL**. A diferencia de las bases de datos relacionales donde hay muchos productos donde elegir, la variedad de Bases de Datos Orientadas a Objetos es mucho menor.

Db4o es una verdadera base de datos de objetos. Muchas otras bases de datos de objetos lo que hacen es utilizar internamente mecanismos relacionales para almacenar los objetos lo que las hace que no sean una verdadera base de datos de objetos (se suelen denominar bases de datos híbridas). El contenido de los objetos, así como la estructura y las relaciones son almacenados en la base de datos sin importar la complejidad que tenga la clase.

Ventaja
Una de las ventajas de db4o es que es un simple jar distribuible con cualquier aplicación sin necesidad de tener que instalar nada. Tampoco necesita *drivers* de tipo JDBC o similar.

Existen distintas distribuciones de la base de datos db4o para Java, .NET y Mono. Cualquier base de datos creada con cualquiera de estos lenguajes es intercambiable entre sí. Eso quiere decir que puedo utilizar una base de datos creada con .NET desde Java y viceversa.

Licencia de db4o

La licencia de db4o al igual que algunas otras herramientas GPL tiene una licencia dual, esto significa que se puede utilizar sin problema para desarrollo, uso interno o asociada a otras herramientas GPL. No obstante, si se quiere utilizar esta herramienta para incorporarla a software comercial se necesitará adquirir una licencia de db4o. Mi consejo es que antes de utilizar cualquier software es preciso conocer el tipo de licencia y si es compatible o no con el uso que le vamos a dar.

Para más información sobre licencias acceder a *http://www.db4o.com/*

Db4o es una base de datos orientada a objetos nativa de Java. La distribución es un único zip y su contenido es el siguiente:

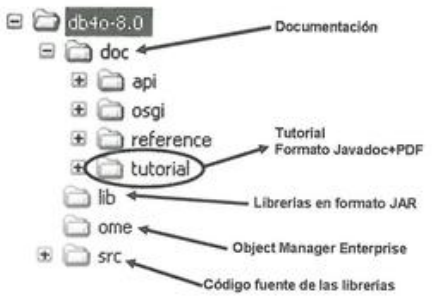


Figura 9.1. Contenido de la distribución db4o versión 8.0

El motor de base de datos

Db4o tiene varias configuraciones para su versión 8.0:

- **db4o-8.0-core*.jar**. Archivos *core* que contienen el núcleo del motor.
- **db4o-8.0-cs*.jar**. Archivos *cs* que contienen la versión cliente/servidor.
- **db4o-8.0-optional*.jar**. Archivos que añaden funcionalidad avanzada al motor de base de datos.
- **db4o-8.0-all*.jar**. Contienen todas las características anteriores. Instalación completa.

La base de datos también está disponible para las diferentes versiones de JDK. Por ejemplo para la instalación completa existen las siguientes versiones:

- **db4o-8.0-all-java1.1.jar**. El cual funciona con la mayoría de JDKs dado que se ha escrito para proporcionar máxima retrocompatibilidad. Sería compatible con los JDK que proporcionen compatibilidad con el JDK 1.1.x.
- **db4o-8.0-java1.2.jar**. Desarrollado para los JDK entre las versiones 1.2 y 1.4.
- **db4o-8.0-java5.jar**. Desarrollado para los JDK 5 y 6.

La instalación consistirá en instalar el motor de base de datos que son las clases necesarias para hacer que funcione la API en toda su extensión y alguna aplicación para visualizar los datos con los que se está trabajando. Sin esta última aplicación se estaría trabajando "a ciegas" con los datos, lo cual no es operativo.

9.3.1 INSTALANDO EL MOTOR DE BASES DE DATOS

La instalación consistirá en colocar solo uno de los ficheros db4o-*.jar dentro del directorio donde apunte la variable CLASSPATH.

Instalación dentro de un IDE
En ocasiones, el programador desea utilizar la base datos en un IDE y situarla dentro de un directorio *./lib* en el directorio de trabajo. En ese caso habrá que configurar el IDE para indicarle que añada la librería (el JAR) db4o situada en el directorio *./lib* al proyecto actual.

El visor de objetos u Object Manager Enterprise
El Object Manager es una herramienta también suministrada en el zip de la instalación. En la distribución se incluye un plugin para el entorno de desarrollo Eclipse que permite ejecutar el *Object Manager*.

9.4 EL API (APPLICATION PROGRAM INTERFACE)

La documentación del API viene en formato JavaDoc en el directorio */doc/api* del zip descargado.

Una de las interfaces más importantes es la siguiente:

`com.db4o.ObjectContainer`

- Este contenedor puede representar una base de datos en modo monousuario (*stand-alone*) o un cliente de un servidor db4o.
- Esta interface proporciona métodos para almacenar, consultar y borrar objetos así como realizar transacciones sobre la base de datos.
- Cada *ObjectContainer* representa una transacción. Todas las operaciones realizadas en la base de datos se hacen en modo transaccional de tal manera que cuando se haga *commit()* o *rollback()*, automáticamente se inicia la siguiente transacción.
- Cada *ObjectContainer* mantiene sus propias referencias a los objetos instanciados y almacenados.

9.5 OPERACIONES BÁSICAS CON LA BASE DE DATOS

Antes de comenzar a trabajar con la base de datos vamos a crear una clase que va a ser la candidata a utilizar cuando interactuemos con la base de datos. Nuestra clase va a ser la siguiente:

```
public class alumno{
    private String nombre;
    private int edad;
    private double nota;
    public alumno(){
          this.nombre = null;
          edad = 0;
          nota = 0;
    }
    public alumno(String n,int e) {
          this.nombre = n;
          this.edad = e;
          this.nota = -1; //nota no establecida
    }
    public alumno(String nom,int e, double not) {
          this.nombre = nom;
          this.edad = e;
          this.nota = not;
    }
    public void setNombre(String n) {
          this.nombre = n;
    }
    public String getNombre() {
          return this.nombre;
    }
    public void setNota(double n) {
          this.nota=n;
    }
    public double getNota() {
          return this.nota;
    }
    public void setEdad(int e) {
          this.edad=e;
    }
    public int getEdad() {
          return this.edad;
    }
    public String toString() {
```

```
            if (this.nota != -1)
                  return this.nombre+"("+this.edad+") Nota:"+this.nota;
            return this.nombre+"("+this.edad+")";
      }
}
```

Como podemos ver la clase anterior almacena los datos de alumnos y tiene los métodos mínimos para poder trabajar con ella. A continuación, vamos a ver las operaciones básicas que se pueden realizar sobre esta base de datos (hablamos de operaciones básicas, en posteriores apartados se profundizará más sobre algunas operaciones ya vistas).

9.5.1 CREAR/ACCEDER A LA BASE DE DATOS

Existe una estructura básica de trabajo con la base de datos la cual es la siguiente:

Apertura de la BD (conexión) → Realizar operaciones → Cerrar la BD (desconexión)

Esta secuencia de trabajo traducida a Java sería la siguiente:

```
ObjectContainer bd = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),"alumnos.
db4o");
try {
//Realizar operaciones o
//llamadas a métodos
}
finally {
bd.close();
}
```

Tabla 9.1. Métodos de apertura y cierre de una base de datos db4o

Acción	Objetivo
Db4oEmbedded.openFile(Db4oEmbedded. newConfiguration(), "alumnos.db4o")	El método *openfile* abre una base de datos y si no existe la crea. Recibe dos parámetros, uno de tipo *Configuration* el cual se puede obtener realizando la llamada *Db4oEmbedded.newConfiguration()*. La interfaz Configuration contiene métodos para poder configurar db4o. El segundo parámetro es el nombre del fichero donde se va a alojar la base de datos.
bd.close()	Se le insta al objeto de tipo *ObjectContainer* a cerrar la base de datos.

Recuerda
Si al abrir la base de datos esta no existe → ésta se creará.
Recuerda que una base de datos solamente se puede abrir una vez. Si se hacen varias aperturas de la base de datos generará una excepción del tipo *DatabaseFileLockedException*.

9.5.2 ALMACENAR OBJETOS

Para almacenar objetos en la base de datos basta con llamar al método *store()* del objeto bd creado de tipo *ObjectContainer*. Un ejemplo de método que tres objetos alumno es el siguiente:

```
public static void almacenarAlumnos(ObjectContainer bd) {
    alumno a1 = new alumno("Juan Gámez",23,8.75);
    bd.store(a1);
    System.out.println(a1.getNombre()+" Almacenado");
    alumno a2 = new alumno("Emilio Anaya",24,6.25);
    bd.store(a2);
    System.out.println(a2.getNombre()+" Almacenado");
    alumno a3 = new alumno("Ángeles Blanco",26,7);
    bd.store(a3);
    System.out.println(a3.getNombre()+" Almacenado");
}
```

Salida:

Juan Gámez Almacenado

Emilio Anaya Almacenado

Ángeles Blanco Almacenado

9.5.3 RECUPERAR OBJETOS DE LA BASE DE DATOS

Cuando estudiamos las bases de datos relacionales las recuperaciones de datos de la base de datos las hacíamos utilizando un lenguaje llamado **SQL** (*Structured Query Language* o lenguaje estructurado de consultas). Este lenguaje es específico para bases de datos relacionales. Con esta Base de Datos Orientada a Objeto utilizaremos otro lenguaje (dado que SQL no está pensado para consultar objetos) llamado **QBE** (Query *By Example*- consulta por ejemplo) el cual es totalmente diferente a SQL al igual que las Bases de Datos Orientadas a Objetos son totalmente diferentes a las relacionales.

Para mostrar el resultado de las consultas realizadas a la base de datos vamos a necesitar un nuevo método *mostrarResultado()* el cual toma como parámetro un *ObjectSet* (conjunto de objetos) y va recorriéndolo mostrando uno a uno los datos recuperados. El código del método es el siguiente:

```
public static void mostrarResultado(ObjectSet  res){
    System.out.println("Recuperados "+res.size()+" Objetos");
    while(res.hasNext()) {
       System.out.println(res.next());
    }
}
```

Imaginemos que queremos recuperar todos los alumnos de nuestra base de datos, para ello se le pasa un resultado vacío (datos de tipo *String* a *null* y campos numéricos a 0) que le indica a la base de datos que deberá de recuperar todos los objetos.

El siguiente código recuperará todos los objetos tipo alumno de la base de datos:

```
public static void muestraAlumnos(ObjectContainer bd) {
    alumno a = new alumno(null, 0, 0);
    ObjectSet res = bd.queryByExample(a);
    mostrarResultado(res);
}
```

El resultado de ejecutar este código es el siguiente:

Recuperados 3 Objetos

Juan Gámez(23) Nota: 8.75

Emilio Anaya(24) Nota: 6.25

Ángeles Blanco(26) Nota: 7

Otras variantes de este tipo de consultas sería el recuperar un alumno concreto por su nombre o un alumno cuya edad sea, por ejemplo, 26. En este caso se le pasa un objeto alumno cuyo campo edad tenga el valor 26:

```
public static void muestraAlumnos26(ObjectContainer bd) {
    alumno a = new alumno(null, 26, 0);
    ObjectSet res = bd.queryByExample(a);
    mostrarResultado(res);
}
```

El resultado de ejecutar este código es el siguiente:

Recuperados 1 Objetos

Ángeles Blanco(26) Nota: 7

9.5.4 ACTUALIZAR OBJETOS EN LA BASE DE DATOS

Actualizar un objeto es muy fácil. Basta con modificarlo y llamar al método *store()* para que se actualice en la base de datos. Veamos un ejemplo:

```
public static void actualizarNotaAlumno(ObjectContainer bd,String nombre,double nota)
{
    ObjectSet res = bd.queryByExample(new alumno(nombre,0,0));
    alumno a = (alumno)res.next();
    a.setNota(nota);
    bd.store(a);
    muestraAlumnos(bd);
}
```

Al realizar la siguiente llamada al método:

```
actualizarNotaAlumno(bd,"Emilio Anaya",9.5);
```

El resultado es el siguiente:

Recuperados 3 Objetos

Juan Gámez(23) Nota: 8.75

Emilio Anaya(24) Nota: 9.5

Ángeles Blanco(26) Nota: 7

Se modificará la nota de Emilio Anaya por el valor 9,5.

Recuerda

Los objetos para poder ser actualizados deben de haber sido insertados o recuperados en la misma sesión, si no se añadirá otro objeto en vez de actualizarse. Db4o para modificar un objeto debe conocerlo previamente.

9.5.5 BORRAR OBJETOS DE LA BASE DE DATOS

El siguiente método borrará un alumno de la base de datos.

```
public static void borrarAlumnoporNombre(ObjectContainer bd,String nombre) {
    ObjectSet res = bd.queryByExample(new alumno(nombre,0,0));
    alumno a = (alumno)res.next();
    bd.delete(a);
    muestraAlumnos(bd);
}
```

Al realizar la siguiente llamada al método:

```
borrarAlumnoporNombre(bd,"Juan Gámez");
```

El resultado es el siguiente:

```
Recuperados 2 Objetos
Emilio Anaya(24) Nota: 9.5
Ángeles Blanco(26) Nota: 7
```


Recuerda

Los objetos para poder ser borrados, al igual que para ser actualizados deben de haber sido insertados o recuperados en la misma sesión. Aunque se proporcione un prototipo con los mismos datos no es suficiente.

9.6 CONSULTANDO LA BASE DE DATOS

Db4o tiene la posibilidad de consultar la base de datos mediante tres tipos de sistemas:

- **Query By Example (QBE)**. Es el sistema ya visto anteriormente.
- **Native Queries (NQ)**. Son consultas nativas. Es la interfaz principal de la base de datos y aconsejado por los desarrolladores de db4o.
- **SODA (Simple Object Data Access)**. Es la API interna. Se puede utilizar para una mayor retrocompatibilidad o para generar consultas dinámicamente. Es mucho más potente que las dos anteriores y mucho más rápida dado que los dos tipos de consultas anteriores (QBE y NQ) tienen que ser traducidas a SODA para ejecutarse.

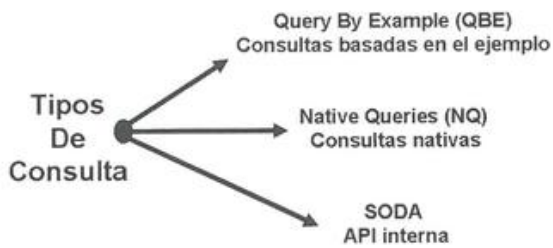


Figura 9.2. Tipos de consultas en db4o

Truco
Si al ejecutar tu programa aparece "AccessibleObject#setAccessible() is not available" probablemente las queries no funcionen con atributos definidos como private. Solución para que funcionen las *queries*: declararlos como públicos.

QBE es la forma más básica de consultar la base de datos. Es sencilla porque funciona presentando un ejemplo y se recuperarán los datos que coincidan con el mismo pero esto hace que tenga muchas limitaciones. Algunas de estas limitaciones al utilizar QBE son las siguientes:

- Al contrario que con otros lenguajes de consulta no se pueden realizar expresiones avanzadas (por ejemplo operadores AND, OR, NOT, etc.).
- Hay que proporcionar un ejemplo con las limitaciones que ello conlleva.
- No se puede preguntar por objetos cuyo valor de un campo numérico sea 0, *Strings* vacíos o algún campo que sea nulo (*null*).
- Se necesita un constructor para crear objetos con campos no inicializados.

9.6.1 LIBRERÍA API SODA

En este apartado vamos a ver como funcionan las consultas utilizando la librería SODA (*Simple Object Database Access*). Esta librería presenta la ventaja que las consultas se ejecutan de la manera más rápida y permite generar una consulta dinámica (las consultas dinámicas son las consultas que solo se conocen en tiempo de ejecución).

Para crear una consulta expresada en SODA se necesita un objeto de tipo *Query* el cual puede ser generado llamando al método *query()* del objeto de tipo *ObjectContainer*. Una vez creado este objeto *Query* se le van añadiendo *Constraints* (restricciones) para ir modelando el resultado que se quiere obtener. Una vez que ya se tiene modelada la consulta se ejecuta la consulta llamando al método *execute()* del objeto tipo *Query*.

Vamos a ver un método el cual genera un listado de todos los objetos alumno de la base de datos.

```
public static void consultaSODAAlumnos(ObjectContainer bd) {
    Query query=bd.query();
    query.constrain(alumno.class);
    ObjectSet result=query.execute();
    mostrarResultado(result);
}
```

El resultado de ejecutar este código es el siguiente:

```
Recuperados 3 Objetos
Juan Gámez(23) Nota: 8.75
```

Emilio Anaya(24) Nota: 6.25

Ángeles Blanco(26) Nota: 7

Si expresamos la consulta anterior de una forma gráfica es como si se construyese un grafo como el siguiente:

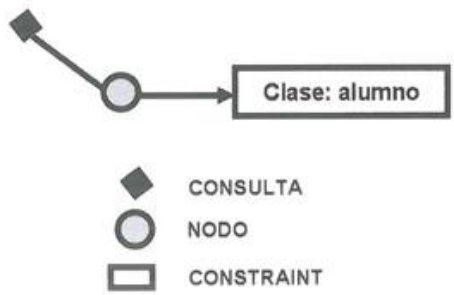


Figura 9.3. Consulta SODA I

Lo que se va a construir es una consulta con una serie de nodos los cuales lo que van a hacer es quedarse o desechar una serie de candidatos respecto a las *constraints* que tengan asociados dichos nodos. Las *constraint* son las que hacen incluir o excluir los posibles candidatos de ese nodo. Los objetos de tipo *Query* son punteros o enlaces a un nodo en SODA.

Tabla 9.2. Objetos Constraint y Query

Objeto	Objetivo
Constraint	Limitar el número de objetos devueltos en la ejecución de una *query*.
Query	Enlace a un nodo en SODA.

Imaginemos que queremos tener todos los alumnos que se llamen "Emilio Anaya", en ese caso deberemos descender al nombre de dicho alumno y añadirle una *constraint* que haga que el nombre sea "Emilio Anaya". El grafo de la consulta que se quiere realizar sería el siguiente:

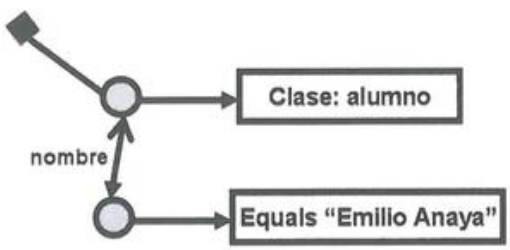


Figura 9.4. Consulta SODA II

```
public static void consultaSODAEmilio(ObjectContainer bd) {
    Query query=bd.query();
    query.constrain(alumno.class);
    query.descend("nombre").constrain("Emilio Anaya");
    ObjectSet result=query.execute();
    mostrarResultado(result);
}
```

El resultado de ejecutar este método será el siguiente:

Recuperados 1 Objetos

Emilio Anaya(24) Nota: 6.25

Algo parecido pasaría si en vez del nombre quisiésemos seleccionar aquellos alumnos que tuviesen 23 años. El grafo sería muy parecido:

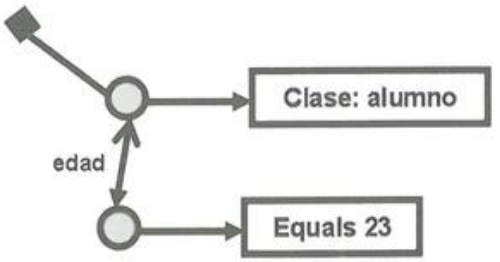


Figura 9.5. Consulta SODA III

El método quedaría ahora modificado de la siguiente manera:

```
public static void consultaSODAAlumVtres(ObjectContainer bd) {
    Query query=bd.query();
    query.constrain(alumno.class);
    query.descend("edad").constrain(23);
    ObjectSet result=query.execute();
    mostrarResultado(result);
}
```

Recuperados 1 Objetos

Juan Gámez(23) Nota: 8.75

Imaginemos ahora que queremos seleccionar los alumnos que no tengan 23 años. El grafo y el código serían los siguientes:

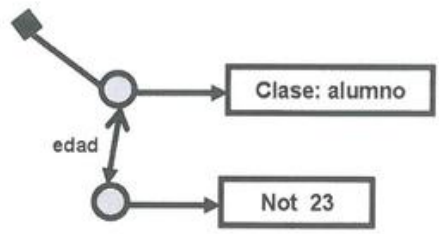


Figura 9.6. Consulta SODA IV

```
Query query=bd.query();
query.constrain(alumno.class);
query.descend("edad").constrain(23).not();
ObjectSet result=query.execute();
mostrarResultado(result);
```

Resultado:

Recuperados 2 Objetos

Emilio Anaya(24) Nota: 6.25

Ángeles Blanco(26) Nota: 7

Imaginemos ahora que queremos conocer los alumnos mayores de 23 años y menores de 25. El grafo y el código serían los siguientes:

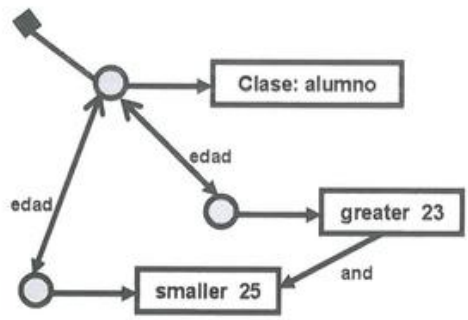


Figura 9.7. Consulta SODA V

```
Query query=bd.query();
query.constrain(alumno.class);
Constraint constr=query.descend("edad").constrain(25).smaller();
```

```
query.descend("edad").constrain(23).greater().and(constr);
ObjectSet result=query.execute();
mostrarResultado(result);
```

Resultado:

Recuperados 1 Objetos

Emilio Anaya(24) Nota: 6.25

Otra consulta podría ser los alumnos cuya nota sea menor de 7 o edad mayor de 25. El grafo y el código de la clase completo serían los siguientes:

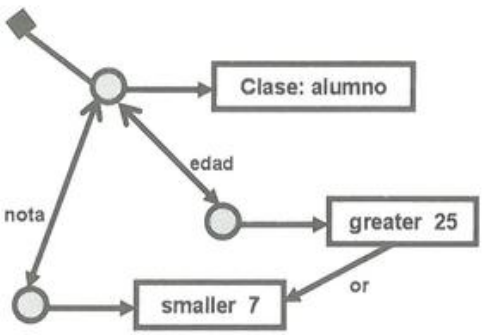


Figura 9.8. Consulta SODA VI

Importante
No olvidar nunca importar las librerías necesarias. En nuestro caso para el siguiente ejemplo serían las siguientes:

- import java.io.*;
- import com.db4o.*;
- import com.db4o.query.*;

```
import java.io.*;
import com.db4o.*;
import com.db4o.query.*;
public class testSODA  {
public static void main(String[] args) {
    new File("alumnos.db4o").delete();
    ObjectContainer bd = Db4oEmbedded.openFile(Db4oEmbedded
    .newConfiguration(), "alumnos.db4o");
    try {
          almacenarAlumnos(bd);
```

```
            consultaSODA(bd);
        } catch(Exception e){
            e.printStackTrace();
        }finally {
            bd.close();
        }
    }

    public static void mostrarResultado(ObjectSet  res){
        System.out.println("Recuperados "+res.size()+" Objetos");
        while(res.hasNext()) {
           System.out.println(res.next());
        }
    }

    public static void almacenarAlumnos(ObjectContainer bd) {
        alumno a1 = new alumno("Juan Gámez",23,8.75);
        bd.store(a1);
        System.out.println(a1.getNombre()+" Almacenado");
        alumno a2 = new alumno("Emilio Anaya",24,6.25);
        bd.store(a2);
        System.out.println(a2.getNombre()+" Almacenado");
        alumno a3 = new alumno("Angeles Blanco",26,7);
        bd.store(a3);
        System.out.println(a3.getNombre()+" Almacenado");
    }

    public static void consultaSODA(ObjectContainer bd) {
        Query query=bd.query();
        query.constrain(alumno.class);
        Constraint constr=query.descend("nota").constrain(7).smaller();
        query.descend("edad").constrain(25).greater().or(constr);
        ObjectSet result=query.execute();
        mostrarResultado(result);
    }

}
```

Resultado:

Recuperados 2 Objetos

Emilio Anaya(24) Nota: 6.25

Ángeles Blanco(26) Nota: 7

Imaginemos que queremos mostrar los alumnos de la base de datos ordenados descendentemente por su edad. El código sería el siguiente:

```
Query query=bd.query();
query.constrain(alumno.class);
query.descend("edad").orderDescending();
ObjectSet result=query.execute();
mostrarResultado(result);
```

El resultado de ejecutar este código es el siguiente:

Recuperados 3 Objetos

Ángeles Blanco(26) Nota: 7

Emilio Anaya(24) Nota: 6.25

Juan Gámez(23) Nota: 8.75

Para hacer la ordenación de manera ascendente añadiríamos la *constraint orderAscending()*.

9.7 TIPOS DE DATOS ESTRUCTURADOS

Llamamos objetos estructurados a aquellos objetos que contienen a su vez otros objetos. Imaginemos que para una asignatura determinada se van a realizar una serie de proyectos. Cada proyecto está asignado a un alumno determinado, por lo tanto, la clase *proyecto* incluirá un objeto de la clase *alumno*. La clase *proyecto* estaría implementada de la siguiente forma:

```
public class proyecto{
    private String descripcion;
    private alumno al;

    public proyecto(String descripcion) {
        this.descripcion=descripcion;
        this.al=null;
    }

    public alumno getAlumno() {
        return al;
    }

    public void setAlumno(alumno a) {
        this.al = a;
    }
```

```
    public String getDescripcion() {
        return descripcion;
    }

    public String toString(){
           return descripcion + "-> "+ al;
    }
}
```

Para insertar los proyectos en la base de datos utilizaremos el siguiente método:

```
public static void almacenarProyectos(ObjectContainer bd) {
    proyecto p1 = new proyecto("Robot con microcontroladores");
    alumno a1 = new alumno("Juan Gámez",23,8.75);
    p1.setAlumno(a1);
    bd.store(p1);
    System.out.println(p1.toString()+" Almacenado");

    proyecto p2 = new proyecto("Webmin sobre Ubuntu");
    alumno a2 = new alumno("Emilio Anaya",24,6.25);
    p2.setAlumno(a2);
    bd.store(p2);
    System.out.println(p2.toString()+" Almacenado");

    proyecto p3 = new proyecto("Joomla y Drupal");
    alumno a3 = new alumno("Ángeles Blanco",26,7);
    p3.setAlumno(a3);
    bd.store(p3);
    System.out.println(p3.toString()+" Almacenado");

    proyecto p4 = new proyecto("Doctor Profiler");
    alumno a4 = new alumno("Luis Alberto García",29,9.5);
    p4.setAlumno(a4);
    bd.store(p4);
    System.out.println(p4.toString()+" Almacenado");
}
```

9.7.1 CONSULTA DE DATOS ESTRUCTURADOS CON SODA

La consulta de datos estructurados con SODA sigue las mismas pautas que utilizamos para las consultas de datos simples. En el siguiente método se van a recuperar aquellos proyectos cuya descripción sea "Doctor Profiler".

```
public static void consultaSODA(ObjectContainer bd) {
    Query query=bd.query();
    query.constrain(proyecto.class);
```

```
        query.descend("descripcion").constrain("Doctor Profiler");
        ObjectSet result=query.execute();
        mostrarResultado(result);
    }
```

9.7.2 CONSULTA DE DATOS ESTRUCTURADOS CON QBE

Vamos a proceder a hacer algunas consultas a nuestra base de datos utilizando QBE. En la primera consulta vamos a mostrar todos los alumnos de la base de datos:

```
public static void muestraAlumnosQBE(ObjectContainer bd) {
    alumno a = new alumno(null, 0, 0);
    ObjectSet res = bd.queryByExample(a);
    mostrarResultado(res);
}
```

Como podemos observar, el código no cambia del anteriormente visto. Aunque el objeto alumno ahora esté dentro del objeto proyecto no será impedimento para poder realizar consultas sobre los alumnos. El resultado de ejecutar este código es el siguiente:

Recuperados 4 Objetos

Juan Gámez(23) Nota:8.75

Emilio Anaya(24) Nota:6.25

Ángeles Blanco(26) Nota:7.0

Luis Alberto García(29) Nota:9.5

Obviamente, si vamos a consultar los proyectos, la consulta tampoco tendrá ninguna dificultad añadida a lo ya visto hasta ahora.

```
public static void muestraProyectosQBE(ObjectContainer bd) {
    proyecto p = new proyecto(null);
    ObjectSet res = bd.queryByExample(p);
    mostrarResultado(res);
}
```

El resultado de ejecutar el código anterior es el siguiente:

Recuperados 4 Objetos

Robot con microcontroladores-> Juan Gámez(23) Nota:8.75

Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25

Joomla y Drupal-> Ángeles Blanco(26) Nota:7.0

Doctor Profiler-> Luis Alberto García(29) Nota:9.5

Una consulta algo más compleja en QBE será el mostrar todos los proyectos en los que trabaje el alumno "Emilio Anaya".

```
public static void muestraProyectoEmilioQBE(ObjectContainer bd) {
    alumno a = new alumno("Emilio Anaya", 0, 0);
    proyecto p = new proyecto(null);
    p.setAlumno(a);
    ObjectSet res = bd.queryByExample(p);
    mostrarResultado(res);
}
```

Como se puede observar tampoco sigue siendo tan complejo. QBE es una forma sencilla aunque con sus limitaciones de consultar a la base de datos. El resultado del código anterior es el siguiente:

Recuperados 1 Objetos

Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25

ACTIVIDADES

- Crea una consulta QBE que muestre los proyectos de aquellos alumnos que tengan 23 años.
- Codifícala en un método al igual que los ejemplos vistos anteriormente y comprueba que funciona.

9.7.3 BORRADO DE DATOS ESTRUCTURADOS

Para demostrar cómo funciona el borrado de datos estructurados echemos un vistazo al siguiente ejemplo:

Truco

Ejecutando el siguiente código:

```
new File("alumnos.db4o").delete();
```

Se borra el fichero que contiene la base de datos antes de comenzar a trabajar con él. De esa manera la base de datos siempre se inicializa al principio de la ejecución y no conserva los datos de ejecuciones anteriores.

```
import java.io.*;
import com.db4o.*;
import com.db4o.query.*;
public class TestBorrado  {
public static void main(String[] args) {
    new File("alumnos.db4o").delete();
    ObjectContainer bd = Db4oEmbedded.openFile(Db4oEmbedded
    .newConfiguration(), "alumnos.db4o");
    try {
          almacenarProyectos(bd);
          borraPorDescripcion(bd);
          mostrarTodosProyectos(bd);
          mostrarTodosAlumnos(bd);
    } catch(Exception e){
          e.printStackTrace();
    }finally {
          bd.close();
    }
}
public static void mostrarResultado(ObjectSet  res){
    System.out.println("Recuperados "+res.size()+" Objetos");
    while(res.hasNext()) {
       System.out.println(res.next());
    }
}

public static void almacenarProyectos(ObjectContainer bd) {
    proyecto p1 = new proyecto("Robot con microcontroladores");
    alumno a1 = new alumno("Juan Gámez",23,8.75);
    p1.setAlumno(a1);
    bd.store(p1);
    System.out.println(p1.toString()+" Almacenado");

    proyecto p2 = new proyecto("Webmin sobre Ubuntu");
    alumno a2 = new alumno("Emilio Anaya",24,6.25);
    p2.setAlumno(a2);
    bd.store(p2);
    System.out.println(p2.toString()+" Almacenado");

    proyecto p3 = new proyecto("Joomla y Drupal");
    alumno a3 = new alumno("Ángeles Blanco",26,7);
    p3.setAlumno(a3);
    bd.store(p3);
    System.out.println(p3.toString()+" Almacenado");

    proyecto p4 = new proyecto("Doctor Profiler");
    alumno a4 = new alumno("Luis Alberto García",29,9.5);
```

```
        p4.setAlumno(a4);
        bd.store(p4);
        System.out.println(p4.toString()+" Almacenado");

    }

    public static void mostrarTodosAlumnos(ObjectContainer bd) {
        Query query=bd.query();
        query.constrain(alumno.class);
        ObjectSet result=query.execute();
        mostrarResultado(result);
    }
    public static void mostrarTodosProyectos(ObjectContainer bd) {
        Query query=bd.query();
        query.constrain(proyecto.class);
        ObjectSet result=query.execute();
        mostrarResultado(result);
    }

    public static void borraPorDescripcion(ObjectContainer bd) {
        Query query=bd.query();
        query.constrain(proyecto.class);
        query.descend("descripcion").constrain("Doctor Profiler");
        ObjectSet result=query.execute();
        while(result.hasNext()) {
               proyecto p= (proyecto)result.next();
               System.out.println("Borrado: "+p);
               bd.delete(p);
        }
    }

    }
```

En el ejemplo anterior se insertan 4 proyectos con sus respectivos alumnos, se borra el proyecto cuya descripción sea "Doctor Profiler" y una vez borrado el proyecto se muestran los proyectos y los alumnos de la base de datos. El resultado de este programa es el siguiente:

Robot con microcontroladores-> Juan Gámez(23) Nota:8.75 Almacenado

Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25 Almacenado

Joomla y Drupal-> Ángeles Blanco(26) Nota:7.0 Almacenado

Doctor Profiler-> Luis Alberto García(29) Nota:9.5 Almacenado

Borrado: Doctor Profiler-> Luis Alberto García(29) Nota:9.5

Recuperados 3 Objetos

Robot con microcontroladores-> Juan Gámez(23) Nota:8.75

Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25

Joomla y Drupal-> Ángeles Blanco(26) Nota:7.0

Recuperados 4 Objetos

Juan Gámez(23) Nota:8.75

Emilio Anaya(24) Nota:6.25

Ángeles Blanco(26) Nota:7.0

Luis Alberto García(29) Nota:9.5

¿Qué es lo que ha ocurrido? Lo que ha ocurrido es que al borrar el proyecto no se ha borrado el alumno asignado al mismo. En cierto sentido es lógico, el borrado de un proyecto no implica que desaparezca el alumno (se le podría asignar otro proyecto).

Si se desea que los borrados de objetos estructurados se realicen en cascada deberemos modificar el código añadiendo la siguiente línea:

```
import com.db4o.config.EmbeddedConfiguration;
```

Y deberemos modificar esta línea:

```
ObjectContainer bd = Db4oEmbedded.openFile(Db4oEmbedded
newConfiguration(), "alumnos.db4o");
```

Por las siguientes:

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().objectClass(proyecto.class).cascadeOnDelete(true);
ObjectContainer bd = Db4oEmbedded.openFile(config, "alumnos.db4o");
```

Como se puede ver en el código se está creando una nueva configuración indicando un borrado en cascada sobre la clase proyecto. Una vez realizado esto el borrado de un proyecto implicará un borrado de los alumnos asociados al mismo.

9.8 ARRAYS DE OBJETOS

Es muy común en cualquier programa el uso de arrays tanto de tipos primitivos como de objetos. Siguiendo con el ejemplo anterior, imaginemos que tenemos una serie de profesores los cuales se encargan de llevar los proyectos de una serie de alumnos. Lo más lógico es implementar la serie de proyectos que un profesor atiende mediante un array de objetos de tipo proyecto. A continuación se va a mostrar la clase profesor:

```
public class profesor{
    private proyecto listaproyectos[];
    private String nombre;
    profesor(proyecto[] l,String n){
          this.nombre = n;
          this.listaproyectos=l;
    }
    public String getNombre(){
          return nombre;
    }
    public proyecto[] getProyectos(){
          return this.listaproyectos;
    }
    public int getNumProyectos(){
          return this.listaproyectos.length;
    }
    public String toString(){
          String str = new String();
          for (proyecto l : listaproyectos){
                str=str+ " *** " +l.toString();
          }
          return nombre + " [[[ "+ str+ " ]]] ";
    }
}
```

Como el alumno habrá podido observar, la clase profesor para que sea totalmente operativa deberá de implementar algún método más. Esto se dejará como ejercicio propuesto para el alumno.

ACTIVIDADES

- Completa la clase profesor con los siguientes métodos: Codifícala en un método al igual que los ejemplos vistos anteriormente y comprueba que funciona.
  - *void setNombre(String)*
  - *String getNombre()*
  - *boolean tieneProyecto(String)*. Devuelve *True* si el profesor lleva un proyecto cuya descripción de proyecto coincide con la pasada como parámetro. *False* en caso contrario.
  - *boolean tieneAlumno(String)*. Devuelve *True* si el profesor lleva a un alumno cuyo nombre coincide con el pasado como parámetro. *False* en caso contrario.
- Añade algún método más que estimes conveniente que tenga la clase.

9.8.1 ALMACENAMIENTO DE OBJETOS Y ARRAYS

El almacenamiento de objetos con arrays sigue la misma filosofía vista hasta ahora. En el siguiente método se puede observar como se crean 5 alumnos, cada uno con su proyecto asignado y luego varios de ellos se asignan a un profesor y el resto a otro profesor.

```
public static void almacenarProfesores(ObjectContainer bd) {
    proyecto p1 = new proyecto("Robot con microcontroladores");
    alumno a1 = new alumno("Juan Gámez",23,8.75);
    p1.setAlumno(a1);
    proyecto p2 = new proyecto("Webmin sobre Ubuntu");
    alumno a2 = new alumno("Emilio Anaya",24,6.25);
    p2.setAlumno(a2);
    proyecto p3 = new proyecto("Joomla y Drupal");
    alumno a3 = new alumno("Ángeles Blanco",26,7);
    p3.setAlumno(a3);
    proyecto p4 = new proyecto("Doctor Profiler");
    alumno a4 = new alumno("Luis Alberto García",29,9.5);
    p4.setAlumno(a4);
    proyecto p5 = new proyecto("Gambas II Linux");
    alumno a5 = new alumno("Robert Sinnock",24,9);
    p5.setAlumno(a5);
    profesor pr1 = new profesor(new proyecto[]{p1,p2,p3},"Juan Carlos");
    bd.store(pr1);
    System.out.println(pr1.toString()+" Almacenado");
    profesor pr2 = new profesor(new proyecto[]{p4,p5},"Emma");
    bd.store(pr2);
    System.out.println(pr2.toString()+" Almacenado");
}
```

Como se puede observar, en la llamada al constructor de los objetos profesor se hace lo siguiente:

```
new proyecto[]{p1,p2,p3}
```

En el código anterior se crea un array de proyectos y se almacenan 3 proyectos en dicho array, los proyectos p1, p2 y p3. Al hacer la llamada de esta forma evitamos tener que crear una variable, asignarle los proyectos y luego utilizarla en la llamada al constructor. Estamos simplificando el método, dejándolo más claro y ahorrando líneas de código.

9.8.2 RECUPERACIÓN DE OBJETOS Y ARRAYS

La recuperación será similar a la ya vista anteriormente. En este punto vamos a mostrar varios métodos y su resultado. En el primer método se quiere mostrar un listado de los profesores y junto con sus datos asociados:

```
public static void muestraProfesores(ObjectContainer bd){
    proyecto p = new proyecto(null);
    profesor pr = new profesor(new proyecto[]{p},null);
```

```
        ObjectSet res = bd.queryByExample(pr);
        mostrarResultado(res);
    }
```

Resultado:

Recuperados 2 Objetos

Juan Carlos [[[*** Robot con microcontroladores-> Juan Gámez(23) Nota:8.75 **

* Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25 *** Joomla y Drupal-> Ángeles Blanco(26) Nota:7.0]]]

Emma [[[*** Doctor Profiler-> Luis Alberto García(29) Nota:9.5 *** Gambas II

Linux-> Robert Sinnock(24) Nota:9.0]]]

En este segundo ejemplo se va a complicar algo la consulta. Se desea recuperar aquellos alumnos que estén realizando un proyecto sobre Joomla y Drupal. El código del método es el siguiente:

```
public static void muestraProfesorDrupal(ObjectContainer bd){
    proyecto p = new proyecto("Joomla y Drupal");
    profesor pr = new profesor(new proyecto[]{p},null);
    ObjectSet res = bd.queryByExample(pr);
    mostrarResultado(res);
}
```

El resultado al ejecutar el método es el siguiente:

Recuperados 1 Objetos

Juan Carlos [[[*** Robot con microcontroladores-> Juan Gámez(23) Nota:8.75 **

* Webmin sobre Ubuntu-> Emilio Anaya(24) Nota:6.25 *** Joomla y Drupal-> Ángeles Blanco(26) Nota:7.0]]]

Como se puede observar, antes se estaba haciendo una consulta sobre un objeto proyecto dentro de un array. En el siguiente caso se va a consultar por un objeto alumno dentro del objeto proyecto. Se necesitará conocer aquellos profesores que tengan un alumno con 29 años.

```
public static void muestraProfesorAlumno29(ObjectContainer bd){
    alumno a = new alumno(null, 29, 0);
    proyecto p = new proyecto(null);
    p.setAlumno(a);
    profesor pr = new profesor(new proyecto[]{p},null);
    ObjectSet res = bd.queryByExample(pr);
    mostrarResultado(res);
}
```

En este caso, el resultado de la consulta es el profesor cuyo alumno tiene 29 años.

```
Recuperados 1 Objetos
Emma [[[ *** Doctor Profiler-> Luis Alberto García(29) Nota:9.5 *** Gambas II
Linux-> Robert Sinnock(24) Nota:9.0 ]]]
```


RESUMEN DEL CAPÍTULO

Una base de datos orientada a objetos es algo muy diferente a una base de datos relacional. Las bases de datos relacionales se han estado utilizando durante mucho tiempo pero dada la creciente utilización de los lenguajes orientados a objetos como Java, una Base de Datos Orientada a Objetos es un mecanismo muy útil en el desarrollo de proyectos.

La consulta a las Bases de Datos Orientadas a Objetos es muy diferente a las relacionales. Las bases de datos relacionales tienen un estándar fuerte como es el SQL mientras que las Bases de Datos OO pueden ser consultadas de diferentes maneras. En el tema se verán distintas formas de consultar la base de datos, teniendo que elegir el alumno la manera más eficiente y cómoda de consultar la BD dependiendo de las características del proyecto.

Una de las características que ofrecen muchas Bases de Datos Orientadas a Objetos es que se integran muy bien con el lenguaje de programación. En el caso de **db4o** se utilizará la API que ofrece para manejar la base de datos.

EJERCICIOS RESUELTOS

1. Realiza un método con el siguiente prototipo:

```
public static void consultaSODA2(ObjectContainer bd)
```

El método deberá de hacer lo siguiente:

- Insertar un nuevo alumno con los siguientes datos:
  - Nombre: "Juan Serrano"
  - Edad: 29
  - Nota: 9.75

- Mostrar los alumnos de la base de datos ordenados por nombre ascendentemente.
- Borrar el alumno "Juan Serrano".

Solución:

```
public static void consultaSODA2(ObjectContainer bd) {
    alumno a=new alumno("Juan Serrano",29,9.75);
    bd.store(a);
    Query query=bd.query();
    query.constrain(alumno.class);
    query.descend("nombre").orderAscending();
    ObjectSet result=query.execute();
    mostrarResultado(result);
    bd.delete(a);
}
```

Salida por pantalla:

Recuperados 4 Objetos

Ángeles Blanco(26) Nota:7.0

Emilio Anaya(24) Nota:6.25

Juan Gámez(23) Nota:8.75

Juan Serrano(29) Nota:9.75

2. Realiza un método con el siguiente prototipo:

```
public static void borraPorEdad(ObjectContainer bd, int edad)
```

Este método deberá de borrar de la base de datos los alumnos que coincidan con la edad indicada en el segundo parámetro. El método además deberá de mostrar la información de los alumnos eliminados.

Solución:

```
public static void borraPorEdad(ObjectContainer bd, int edad) {
    Query query=bd.query();
    query.constrain(alumno.class);
    query.descend("edad").constrain(edad);
    ObjectSet result=query.execute();
    while(result.hasNext()) {
          alumno a= (alumno)result.next();
          System.out.println("Borrado: "+a);
          bd.delete(a);
    }
}
```

- **3.** Realiza un método con el siguiente prototipo:

```
public static void muestraProyectoNotaAltaQBE(ObjectContainer bd)
```

Este método deberá consultar en la base de datos de alumnos vista en este capítulo aquellos proyectos en los que esté trabajando el alumno con mayor nota de la clase (la nota más alta es un 9,5).

Solución:

```
public static void muestraProyectoNotaAltaQBE(ObjectContainer bd) {
    alumno a = new alumno(null, 0, 9.5);
    proyecto p = new proyecto(null);
    p.setAlumno(a);
    ObjectSet res = bd.queryByExample(p);
    mostrarResultado(res);
}
```

Salida por pantalla:

Recuperados 1 Objetos

Doctor Profiler-> Luis Alberto García(29) Nota:9.5

EJERCICIOS PROPUESTOS

- **1.** Realiza una consulta a la base de datos de alumnos del ejemplo visto en el tema y selecciona aquellos alumnos que no se llamen "Fernando Gil" y ordénalos ascendentemente. Reutiliza el código visto en el tema.

- **2.** Realiza una consulta a la base de datos de alumnos del ejemplo visto en el tema y selecciona aquellos alumnos que no se llamen "Andrés Rosique" o "Juan Gámez". Reutiliza el código visto en el tema.

- **3.** Realiza una consulta a la base de datos de alumnos del ejemplo visto en el tema y selecciona aquellos alumnos cuya nota este entre 7 y 9. Reutiliza el código visto en el tema.

- **4.** (Ejercicio de dificultad alta) Se propone al alumno que cree un juego de preguntas. Para ello se creará una clase pregunta con una pregunta a realizar, cuatro posibles respuestas y una única respuesta verdadera. El alumno deberá crear un fichero de texto con una serie de preguntas. Cada pregunta ocupará 5 líneas del fichero de texto. La forma de guardar los datos en el fichero será la siguiente:
  - pregunta.
  - respuesta 1.
  - respuesta 2.

- respuesta 3.
- respuesta 4.
- solución.

A continuación, se muestra un ejemplo de contenido del fichero con dos preguntas:

```
¿Cuál es el inventor del lenguaje Java?
Steve Jobs
James Gosling
Bill Gates
Andrew Tanenbaum
2

¿Quién es el considerado "padre" de la Web?
Sergey Brin
Joe Cocker
Tim Berners-Lee
Larry Page
3
```

El programa deberá leer el fichero, crear los objetos de tipo pregunta y almacenarlos en la base de datos.

También deberá de tener un método que realice 10 preguntas al usuario y muestre un sumario a la conclusión con las preguntas acertadas y erradas.

- **5.** Modifica el programa anterior para que al usuario se le hagan preguntas de forma aleatoria y sin repetición y que el número de preguntas a realizar pueda ser establecido previamente antes de iniciarse la batería de preguntas.

- **6.** Modifica el programa anterior creando una clase categoría en la que se puedan clasificar las preguntas. Cada categoría tendrá una serie de preguntas almacenadas en un array. El usuario, cuando ejecute el programa, deberá de elegir la categoría de la que quiere ser preguntado.

- **7.** Añádele una interfaz gráfica al programa anterior. El diseño de la interfaz se dejará a criterio del alumno. La interfaz deberá contemplar todas las características del diseño propuesto. Mejora la interfaz añadiendo el campo *jugador*. Cuando el jugador ingresa en el programa debe introducir su nombre y posteriormente jugar. Crea una opción en el programa en la que se muestre el jugador que ostenta el record del juego.

- **8.** Crea otro programa que permita generar las preguntas del archivo de datos. El programa deberá de tener una interfaz gráfica. El diseño de la interfaz lo realizará el alumno atendiendo las necesidades del programa.

- **9.** (Ejercicio de dificultad alta) Necesitamos crear una pequeña aplicación que gestione el parque de ordenadores de una pequeña empresa. Cada ordenador tendrá las siguientes características:
  - Marca.
  - Modelo.
  - Procesador.

- TipoMemoria.
- CantidadMemoria.
- Ubicación.
- Número Serie.
- Número interno (código suministrado por la empresa).

El programa deberá poder Insertar, Modificar y Eliminar equipos. Cuando se inserta un equipo el sistema deberá de dotarle de un código interno.

Se deberán de poder realizar consultas como:

- Listado de equipos de una ubicación determinada.
- Cuántos equipos hay por ubicación.
- Tipo de memoria y cantidad de los equipos.

El diseño de la interfaz lo realizará el alumno atendiendo las necesidades del programa.

Direcciones de interés

- *www.wikipedia.org*

  La enciclopedia libre.

- *http://www.chuidiang.com/java/index.php*

  Página web donde encontrar tutoriales y muchos ejemplos en el lenguaje Java para principiantes y usuarios más avanzados.

- *http://www.programacion.com/java*

  Página web con muchos artículos interesantes y código sobre Java.

- *http://download.oracle.com/javase/*

  Portal donde está toda la documentación sobre la plataforma Java Standar Edition. La sección de tutoriales es magnífica. Aquí el alumno puede encontrar tutoriales muy completos sobre todo tipo de recursos Java.

Material adicional

El material adicional de este libro puede descargarlo en nuestro portal Web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

Índice Alfabético