PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS

JUAN CARLOS MORENO PÉREZ

PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS
© Juan Carlos Moreno Pérez

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-300-7
Todos los derechos reservados © RA-MA, S.A. Editorial y Publicaciones, Madrid, España.

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

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

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-455-4

E-Book desarrollado en España en septiembre de 2014

Dedicado a Aida Ruiz Sánchez.

Índice

INTRODUCCIÓN **11**

CAPÍTULO 1. METODOLOGÍA DE LA PROGRAMACIÓN **13**
1.1 PROGRAMACIÓN ESTRUCTURADA 15
1.1.1 La estructura básica de un programa 16
1.2 LOS DATOS 17
1.2.1 Tipos de datos simples 17
1.2.2 Constantes y literales 19
1.2.3 Variables 20
1.3 OPERADORES Y EXPRESIONES 22
1.3.1 Operadores aritméticos 22
1.3.2 Operadores relacionales 23
1.3.3 Operadores lógicos 24
1.3.4 Operadores unitarios o unarios 24
1.3.5 Operadores de bits 25
1.3.6 Operadores de asignación 25
1.3.7 Precedencia de operadores 26
1.4 ESTRUCTURAS BÁSICAS (SECUENCIAL, CONDICIONAL, ITERATIVA) 27
1.4.1 Estructura secuencial 27
1.4.2 Estructura condicional 28
1.4.3 Estructura iterativa 30
1.4.4 Otros tipos de estructuras 33
1.5 MÉTODOS PARA LA ELABORACIÓN DE ALGORITMOS 35
1.6 RECURSIVIDAD 36

CAPÍTULO 2. ESTRUCTURA DE DATOS **39**
2.1 ESTRUCTURAS ESTÁTICAS 40
2.1.1 Arrays o vectores 40
2.1.2 Arrays multidimensionales o matrices 44
2.1.3 Las cadenas de caracteres 44
2.2 ESTRUCTURAS DINÁMICAS 50
2.2.1 Pilas 53
2.2.2 Colas 56
2.3 TIPOS ABSTRACTOS DE DATOS 61

CAPÍTULO 3. PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS....63
3.1 EL ENTORNO DE DESARROLLO DE PROGRAMACIÓN 64
3.1.1 ¿Es necesario un IDE para compilar y ejecutar Java? 66
3.2 HERRAMIENTAS DE DEPURACIÓN 67
3.3 LA REUTILIZACIÓN DEL SOFTWARE 67
3.4 HERRAMIENTAS DE CONTROL DE VERSIONES 69
3.4.1 ¿Cómo se almacenan las versiones?....69
3.4.2 ¿Cómo se colabora en un sistema de control de versiones?....70
3.4.3 ¿Cómo se trabaja en un sistema de control de versiones?....70
3.4.4 Sistemas de control de versiones centralizados: el repositorio....72
3.4.5 Sistemas de repositorio. Modelos de versionado 73
3.4.6 Apache Subversion....74

CAPÍTULO 4. INTERFACES Y ENTORNOS GRÁFICOS....77
4.1 CARACTERÍSTICAS DE LAS INTERFACES, INTERACCIÓN HOMBRE-MÁQUINA....78
4.2 DISEÑO DE INTERFACES....79
4.3 INTERFACES GRÁFICAS DE USUARIO. CREACIÓN DE NUESTRA PRIMERA APLICACIÓN CON SWING....79
4.3.1 Los componentes Swing. Librerías 81
4.3.2 Los contenedores Swing....82
4.3.3 Organización de los controles en un contenedor 82
4.3.4 Apariencia de las ventanas 84
4.4 PROGRAMACIÓN POR EVENTOS. CONCEPTO DE EVENTOS Y CONTROLADOR DE EVENTOS 85
4.5 GENERACIÓN DE PROGRAMAS EN ENTORNO GRÁFICO 89
4.6 TÉCNICAS DE USABILIDAD....93
4.6.1 La simplicidad como bandera de la usabilidad 94
4.6.2 Algunos consejos a la hora de diseñar un interfaz. Rendimiento del interfaz 94

CAPÍTULO 5. ACCESO A BASES DE DATOS Y OTRAS ESTRUCTURAS....99
5.1 OBJETOS DE LA BASES DE DATOS. LA ARQUITECTURA JDBC....101
5.1.1 ¿Qué se necesita para trabajar con bases de datos y JDBC? 102
5.2 CONEXIONES PARA EL ACCESO A DATOS....103
5.3 MANEJANDO SQLEXCEPTIONS....104
5.4 CREACIÓN Y CARGA DE DATOS EN TABLAS....105
5.4.1 Creación de tablas con JDBC....106
5.4.2 Carga de datos en las tablas con JDBC 108
5.5 RECUPERACIÓN DE LA INFORMACIÓN DE LA BASE DE DATOS 110
5.5.1 La interfaz Resultset....111
5.5.2 Otra manera de recuperar los datos de una tabla....112
5.5.3 Los cursores 113

5.6 MODIFICACIÓN Y ACTUALIZACIÓN DE LA BASE DE DATOS 114
5.6.1 Modificación clásica de datos 114
5.6.2 Modificación de datos en las tablas utilizando Resultset 114
5.6.3 Insertar datos en las tablas utilizando Resultset 115

CAPÍTULO 6. PRUEBAS 119
6.1 OBJETIVOS DE LAS PRUEBAS 120
6.2 TIPOS DE PRUEBAS 121
6.3 PLANIFICACIÓN DE LAS PRUEBAS 123
6.4 PROCESO DE PRUEBAS Y DOCUMENTACIÓN DE LAS MISMAS 123
6.4.1 Planificación de las pruebas: el plan de pruebas 123
6.4.2 Preparación de los datos de prueba 124
6.4.3 Codificación de las pruebas 125
6.4.4 Ejecución de las pruebas 125
6.4.5 Generación del informe final de las pruebas 126
6.5 PRUEBAS DE RENDIMIENTO 126
6.6 NORMAS DE CALIDAD 128

CAPÍTULO 7. HERRAMIENTAS DE GENERACIÓN DE PAQUETES 131
7.1 LOS FICHEROS JAR: FUNCIONES Y CARACTERÍSTICAS 132
7.1.1 Crear un fichero JAR 132
7.1.2 Ver el contenido del JAR 133
7.1.3 Extraer los ficheros de un JAR 133
7.1.4 Ejecutar la aplicación contenida en un JAR 133
7.1.5 ¿Qué es el manifest o manifiesto de un JAR? 134
7.1.6 Problemas con los ficheros JAR 134
7.2 OTROS EMPAQUETADORES: EMPAQUETAMIENTO, INSTALACIÓN Y DESPLIEGUE 134
7.2.1 Wrappers 134
7.2.2 Los instaladores 137
7.2.3 JWS 138

CAPÍTULO 8. DOCUMENTACIÓN DE APLICACIONES 141
8.1 HERRAMIENTAS DE DOCUMENTACIÓN: CARACTERÍSTICAS 142
8.2 DOCUMENTACIÓN DE UNA APLICACIÓN 144

SOLUCIONARIO DE LOS TEST DE CONOCIMIENTOS 149

MATERIAL ADICIONAL 151

ÍNDICE ALFABÉTICO 153

Introducción

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

He intentado que la estructura del libro sea lo más didáctica posible. Los conceptos en el libro son fáciles de comprender, acompañados de muchas fotos, ejemplos y explicaciones sencillas. En los contenidos teóricos, se han intentado exponer los conceptos de una forma simplificada, incluyendo siempre los más importantes. Para asimilar todos los conceptos de cada capítulo, he utilizado muchos programas, notas, consejos, etc. De esa manera y con paciencia aprenderás a programar de forma estructurada sin problemas.

El libro trata conceptos muy interesantes como la programación estructurada, el desarrollo de interfaces, programación con estructuras dinámicas, las pruebas, la documentación, el acceso a bases de datos, herramientas de generación de paquetes, etc. Para el desarrollo del libro he elegido como lenguaje de programación Java. Java es un lenguaje con el cual vas a poder trabajar de forma estructurada y como es multiplataforma no importa que utilices Linux, Windows, Mac OS X u otro sistema, podrás desarrollar, compilar y ejecutar tus programas en cualquier sistema operativo.

El lector, para dominar la materia, además de manejar el libro, deberá 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.

1 METODOLOGÍA DE LA 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 y que luego traduciremos al lenguaje máquina, entendible por los ordenadores, mediante otros programas llamados **intérpretes** o **compiladores**.

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

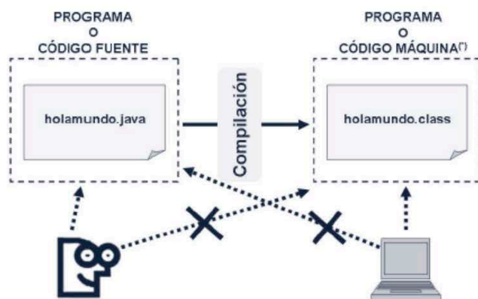

Figura 1.1. *Proceso de compilación en Java*

Como se puede observar, el **código fuente** es el que escribe el programador y 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**).

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

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

El éxito de **Java** 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.

1.1 PROGRAMACIÓN ESTRUCTURADA

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

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

Los programas en Java
Los programas o aplicaciones en Java se componen de 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 o procedimiento principal main() situado en una clase o fichero principal.

1.1.1 LA ESTRUCTURA BÁSICA DE UN PROGRAMA

En este apartado se va a trabajar con el programa de inicio por excelencia en cualquier lenguaje de programación ("*Hola mundo*") y se comentarán cada una de sus líneas.

```
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 órdenes en **Java** terminan en ";" salvo los cierres de llaves a los cuales no hace falta ponérselo, pues se sobreentiende que finaliza la orden).

1.2 LOS DATOS

En este apartado estudiaremos en profundidad los datos que maneja un programa (tipos de datos **simples**, **constantes**, **literales** y **variables**).

1.2.1 TIPOS DE DATOS SIMPLES

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

Tipo de datos	Información representada	Rango	Descripción
byte	Datos enteros	-128 ←→ +127	Se utilizan 8 bits (1 byte) para almacenar el dato.
short	Datos enteros	-32768 ←→ +32767	Dato de 16 bits de longitud (independientemente de la plataforma).
int	Datos enteros	-2147483648 ←→ +2147483647	Dato de 32 bits de longitud (independientemente de la plataforma).
long	Datos enteros	-9223372036854775808 ←→ +9223372036854775807	Dato de 64 bits de longitud (independientemente de la plataforma).
char	Datos enteros y caracteres	0 ←→ 65535	Este rango es para representar números en unicode, los ASCII se representan con los valores del 0 al 127. ASCII es un subconjunto del juego de caracteres Unicode.
float	Datos en coma flotante de 32 bits	Precisión aproximada de 7 dígitos	Dato en coma flotante de 32 bits en formato IEEE 754 (1 bit de signo, 8 para el exponente y 24 para la mantisa).
double	Datos en coma flotante de 64 bits	Precisión aproximada de 16 dígitos	Dato en coma flotante de 64 bits en formato IEEE 754 (1 bit de signo, 11 para el exponente y 52 para la mantisa).
boolean	Valores booleanos	true/false	Utilizado para evaluar si el resultado de una expresión booleanas es verdadero (true) o falso(false).

Tabla 1.1. *Tipos de datos simples*

ACTIVIDADES

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

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

Tipo de dato	Código
byte	byte a;
short	short b, c=3;
int	int d = -30; int e = 0xC125;
long	long b=434123; long b=5L; /* la L en este caso indica Long*/
char	char car1='c'; char car2=99; /*car1 y car2 son lo mismo porque el 99 en ASCII es la 'c' */
float	float pi=3.1416; float pi=3.1416F; /* la F en este caso indica Float*/ float medio=1/2F; /*0.5*/
double	double millón=1e6; /* 1x106 */ double medio=1/2D; /*0.5 la D en este caso indica Double*/
boolean	boolean adivinanza=true;

Tabla 1.2. *Utilización de tipos de datos*

1.2.2 CONSTANTES Y LITERALES

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).

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

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

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

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

RECUERDA

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

Los literales

Un **literal** puede ser una expresión:

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

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

1.2.3 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.

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

RECUERDA

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

Visibilidad y vida de las variables

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

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

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

1.3 OPERADORES Y EXPRESIONES

Todos los lenguajes de programación tienen operadores incluso algunos tienen precedencia sobre otros. En este apartado se estudiarán en profundidad los operadores y expresiones.

1.3.1 OPERADORES ARITMÉTICOS

Los **operadores aritméticos** son utilizados para realizar operaciones matemáticas.

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

Tabla 1.3. *Operadores aritméticos*

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

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

1.3.2 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:

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

Tabla 1.4. *Operadores relacionales*

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

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

1.3.3 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:

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

Tabla 1.5. *Operadores lógicos*

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

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

1.3.4 OPERADORES UNITARIOS O UNARIOS

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

Tabla 1.6. *Operadores unitarios*

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

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

1.3.5 OPERADORES DE BITS

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

Tabla 1.7. *Operadores de bits*

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

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

1.3.6 OPERADORES DE ASIGNACIÓN

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

Tabla 1.8. *Operadores de asignación*

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

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

1.3.7 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 figura:

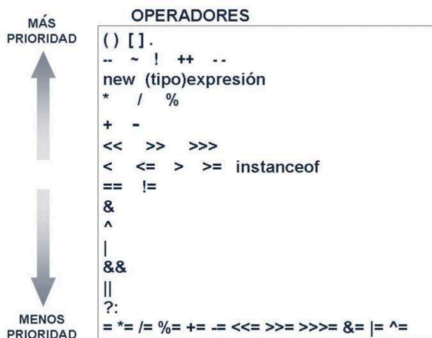

Figura 1.2. *Precedencia de operadores*

Imaginemos que se tiene un código como el siguiente:

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

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

1.4 ESTRUCTURAS BÁSICAS (SECUENCIAL, CONDICIONAL, ITERATIVA)

IMPORTANTE

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

1.4.1 ESTRUCTURA SECUENCIAL

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

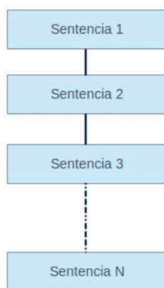

Figura 1.3. *Diagrama de flujo de una estructura secuencial*

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

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

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

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

1.4.2 ESTRUCTURA CONDICIONAL

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

Las 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:

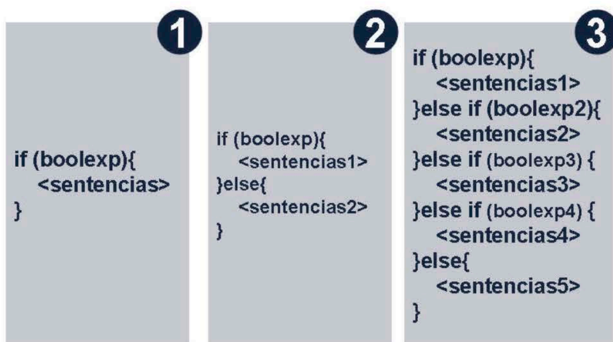

Figura 1.4. *Diferentes sentencias if*

En los casos anteriores, **boolexp** es una expresión booleana que puede ser verdadera o falsa, ejemplos de expresiones booleanas pueden ser: (a > 20 o 2*PI*radio>30). Dependiendo 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:

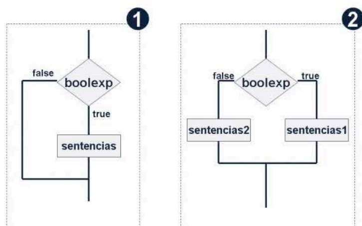

Figura 1.5. *Diagrama de flujo de las sentencias if*

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

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

Otro ejemplo anidando las sentencias *if*:

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

Las estructuras switch

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

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

Figura 1.6. *La sentencia switch*

RECUERDA

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

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

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

1.4.3 ESTRUCTURA ITERATIVA

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

CUIDADO

Ten cuidado con los bucles infinitos. Los bucles infinitos son aquellos que no terminan nunca (aquellos cuya expresión 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.

La estructura iterativa while

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

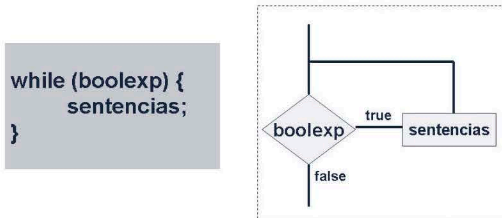

Figura 1.7. *Estructura del bucle while*

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.

La estructura iterativa do while

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

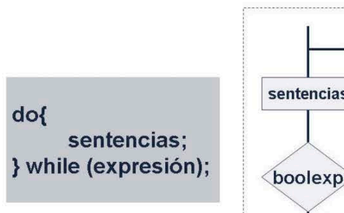

Figura 1.8. *Estructura del bucle do while*

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*.

La estructura iterativa for

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

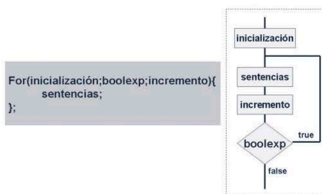

Figura 1.9. *Estructura del bucle for*

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.

1.4.4 OTROS TIPOS DE ESTRUCTURAS

CONSEJO

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

La sentencia break

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

La sentencia return

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

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

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:

```
try {
    Sentencias a proteger
}catch (excepción_1){
    Control de la excepción 1
}
...
catch (excepción_n){
    Control de la excepción n
}[finally{
    Control opcional
}]
```

Figura 1.10 *Estructura del control de excepciones en Java*

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

1.5 MÉTODOS PARA LA ELABORACIÓN DE ALGORITMOS

En primer lugar vamos a conocer qué es un **algoritmo**. La palabra algoritmo viene de un árabe llamado **Muhammad ibn Musa Al-Khowarizmi** que en el siglo IX antes de Cristo acompañaba sus soluciones algebraicas con demostraciones geométricas.

El objetivo de un **algoritmo** es seguir una serie de pasos para al final del mismo conocer la solución a un determinado problema. Para la resolución del problema o explicación del algoritmo se utiliza un **lenguaje algorítmico** que bien puede ser gráfico o bien puede ser no gráfico (en este caso se le suele denominar pseudocódigo).

Veamos cómo se resolvería un problema concreto. Se pide la edad a dos sujetos y el algoritmo determina el sujeto A es mayor que el sujeto B.

Figura 1.11. *Resolución de un programa mediante notación gráfica y pseudocódigo*

Como se puede ver, para la resolución gráfica se utilizan algunos símbolos conocidos para hacer diagramas, que puedes encontrar en versiones de **office**, **libreoffice** y otros programas más específicos para estos menesteres.

En cuanto al **pseudocódigo**, generalmente se pueden utilizar palabras en inglés o simplemente en español. Esto queda a tu gusto. Lo que es importante es que el proceso describa paso a paso de forma inequívoca la resolución de un problema concreto.

Una vez escrito el algoritmo de forma gráfica o en notación de pseudocódigo, ya estamos en disposición de traducirlo a un lenguaje de programación sin problemas. Dependiendo de lo grande del algoritmo, se suele utilizar notación gráfica o pseudocódigo. Para algoritmos complejos, largos o con muchas estructuras condicionales e iterativas la mejor solución es el pseudocódigo porque es más legible. La notación gráfica en estos casos no es posible dado que se crean diagramas muy grandes.

1.6 RECURSIVIDAD

La **recursividad** es una forma de expresar un algoritmo en el que en la solución del problema, el algoritmo se hace una llamada a sí mismo. Generalmente se utiliza la recursividad para resolver problemas que son recursivos como el factorial, la sucesión de Fibonacci, etc. La llamada de un algoritmo a sí mismo se llama **recursión** o **llamada recursiva**.

Entre las ventajas de la recursividad está el que los algoritmos son más reducidos que en una solución iterativa, son más fáciles de leer e interpretar, pero por contra consumen muchos más recursos de máquina dada su naturaleza. Son menos eficientes en cuestión de rendimiento.

Veamos el pseudocódigo de la resolución del factorial de un número mediante un algoritmo recursivo:

```
FUNCIÓN Factorial(NUM)
    RESULTADO: Entero

    SI (NUM<2) ENTONCES
        RESULTADO = 1;
    SINO
        RESULTADO = NUM * Factorial(NUM-1);
    FSI

    DEVOLVER RESULTADO;
FIN FUNCIÓN
```

Como se puede ver en el pseudocódigo anterior, la resolución del factorial es 1 si el número es menor que dos, en caso contrario el resultado será **NUM** por el factorial de **NUM -1**.

TEST DE CONOCIMIENTOS

1 La programación estructurada surgió...

a) en la década de los 60
b) en la década de los 70
c) en la década de los 80

2 ¿Cuál de las siguientes sentencias es correcta?:

a) private static void main (String [ ] args)
b) public static void main (String [ ] args)
c) public void main (String [ ] args)

3 ¿Cuál de las siguientes sentencias es incorrecta?:

a) int e = 0xC125;
b) char car2=99;
c) float pi=3.1416F;
d) double millón=1exp6;

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Bytecode es binario, lenguaje directamente ejecutable por la máquina.
b) Se desaconseja el uso de las sentencias *break* salvo para la estructura *switch*.
c) En **Java**, la clase cuyo nombre coincide con el nombre del fichero debería de llevar el modificador *public*.
d) Los programas o aplicaciones en **Java** se componen de una serie de ficheros **class** que son ficheros en **bytecode**.

5 ¿Cuál de las siguientes afirmaciones es falsa?:

a) En **Java** generalmente cada clase es un fichero distinto.
b) Este comentario /** comentario */ no es correcto puesto que comienza con dos asteriscos.
c) Una variable es una zona de memoria donde se puede almacenar información.
d) El **método main**() al ser estático (static) se le puede llamar sin tener que instanciar la clase.

6 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Los *float* tienen una precisión aproximada de 7 dígitos.
b) Las variables miembros de una clase se inicializan por defecto.
c) Visibilidad, *scope* o ámbito de una variable no son sinónimos.
d) La estructura iterativa *do while* es una variante del *while*.

2 ESTRUCTURA DE DATOS

En el capítulo anterior ya vimos cómo trabajar con datos simples. En este tema vamos a abordar las **estructuras de datos**. Con las estructuras de datos podemos organizar conjuntos de datos elementales de tal manera que es más fácil trabajar con ellos y se pueden resolver problemas que con datos simples costaría mucho más realizarlos.

Las **estructuras de datos** se pueden clasificar en dos grupos, las **estructuras estáticas** y las **estructuras dinámicas** las cuales pueden llegar a ser más complejas y se hace uso de punteros. Generalmente las estructuras de datos suelen tener asociadas operaciones como altas, bajas, búsquedas, ordenamiento, etc., lo que las hace muy potentes y muy eficientes al programar.

Dependiendo del problema a resolver, intentaremos utilizar si es necesario la estructura de datos más apropiada para ello. Existen muchas estructuras de datos como pueden ser listas, conjuntos, matrices, árboles, grafos, *heaps*, tablas y *hash*. En este tema vamos a estudiar en **profundidad las estructuras estáticas y dinámicas** más sencillas y más útiles.

RECUERDA

Un puntero es una **variable** que apunta a una dirección de memoria donde reside un dato o una estructura de datos.

2.1 ESTRUCTURAS ESTÁTICAS

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

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

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

2.1.1 ARRAYS O VECTORES

Hasta ahora, en todos los ejercicios que se han visto durante el capítulo anterior no había una necesidad muy grande de almacenar muchos valores, se utilizaban variables simples en las que podemos 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.

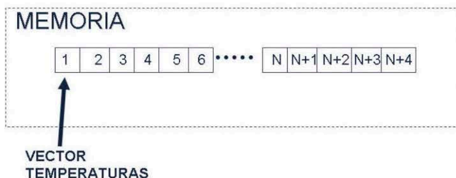

Figura 2.1. *Almacenamiento del vector temperaturas en memoria*

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

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.

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 de código del ejemplo anterior en una sola.

Inicialización de vectores

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

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

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

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

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*.

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.

2.1.2 ARRAYS MULTIDIMENSIONALES O MATRICES

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

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

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

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

En el ejemplo anterior se ha creado un *array* de 2 filas y 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 2.2. *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;
        }
}
```

2.1.3 LAS CADENAS DE CARACTERES

RECUERDA

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

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

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

En el ejemplo anterior las variables nombre1 y nombre2 contienen exactamente lo mismo dado que internamente **Java** almacena los caracteres con sus símbolos **ASCII** correspondientes (a la 'p' le corresponde el 112 y a la 'e' el 101). La variable 3 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'.

¿SABÍAS QUE?

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.

La clase String

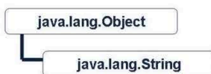

Figura 2.3. *La **clase** string desciende de Object*

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

¿SABÍAS QUE?

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

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

String(String dato)

Constructor de la clase **String**

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

Las 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: "Emma".

int length()

Muestra la longitud de un objeto de la clase **String**

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

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

String concat(String s)

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

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

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

String toString()

Devuelve el propio String

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

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

int compareTo(String s)

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

< 0 Si es menor el *string* desde el que se hace la llamada al String pasado como parámetro.

= 0 Si es igual el *string* desde el que se hace la llamada al String pasado como parámetro.

> 0 Si es mayor el *string* desde el que se hace la llamada al String pasado como parámetro.

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

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

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

CUIDADO

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

boolean equals()

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

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

El código anterior mostrará por pantalla "SON IGUALES".

String trim()

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

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

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

String toLowerCase()

Convierte las letras mayúsculas del objeto String en minúsculas.

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

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

String toUpperCase()

Convierte las letras minúsculas del objeto String en mayúsculas.

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

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

String replace(char car, char newcar)

Reemplaza cada ocurrencia del carácter car por el carácter newcar.

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

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

String substring(int i, int f)

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

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

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

boolean startsWith(String cad)

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

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

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

boolean endsWith(String cad)

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

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

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

char charAt(int pos)

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

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

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

CUIDADO

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

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

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

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

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

char[] toCharArray()

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

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

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

String valueOf(int dato)

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

```
int edad1=6;
String str=String.valueOf(edad1);
float edad2=6;
str=String.valueOf(edad2);
long edad3=6;
str=String.valueOf(edad3);
double edad4=6.5;
str=String.valueOf(edad4);
```

¿Cómo convertir un string en un número?

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

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

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

2.2 ESTRUCTURAS DINÁMICAS

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

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

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

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

Visualmente un nodo sería una cosa parecida a esta:

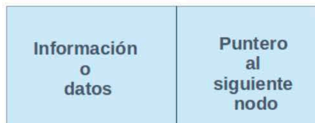

Figura 2.4. *Estructura de un nodo*

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

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

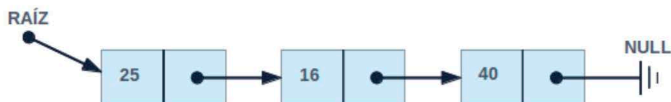

Figura 2.5. *Estructura dinámica de datos*

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

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

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

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

Gráficamente dicha inicialización quedaría de la siguiente forma:

Figura 2.6 *Inicialización de la raíz*

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

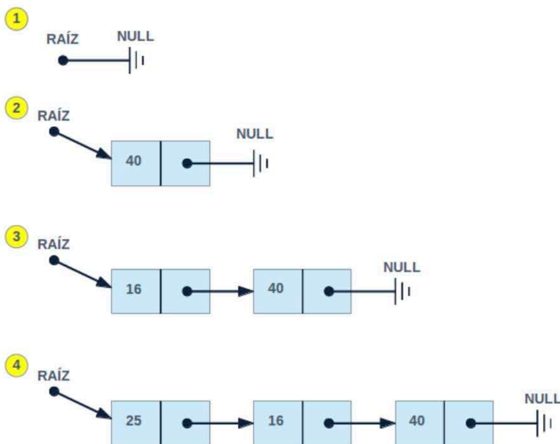

Figura 2.7 *Proceso de creación de una estructura dinámica*

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

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

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

RECUERDA

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

A continuación se estudiarán en profundidad algunas de estas estructuras:

2.2.1 PILAS

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

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

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

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

Caso 1: en el primer caso la pila está vacía y por lo tanto apunta a *null* (**raiz==null**).

Caso 2: en el segundo caso, la pila **al menos tiene un elemento**.

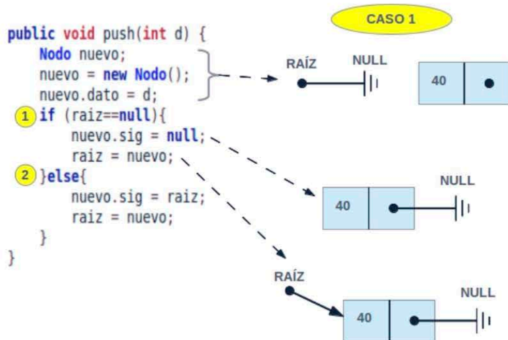

Figura 2.8. *Caso 1 de la función push*

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

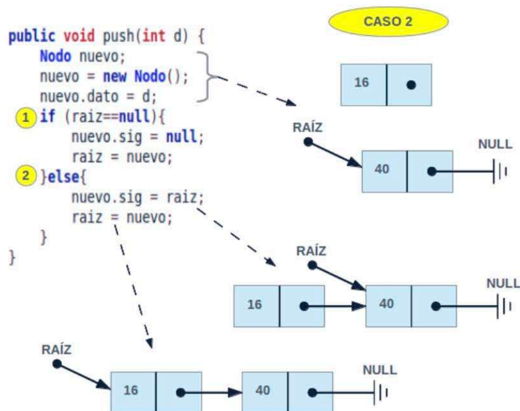

Figura 2.9. *Caso 2 de la función push*

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

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

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

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

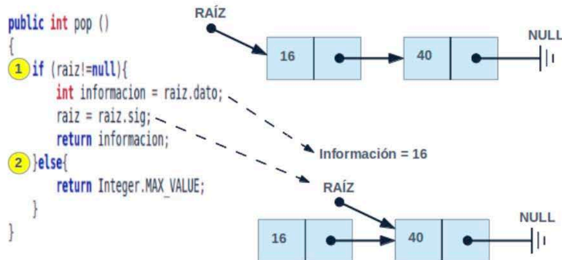

Figura 2.10. *Función pop*

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

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

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

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

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

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

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

ACTIVIDADES

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

2.2.2 COLAS

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

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

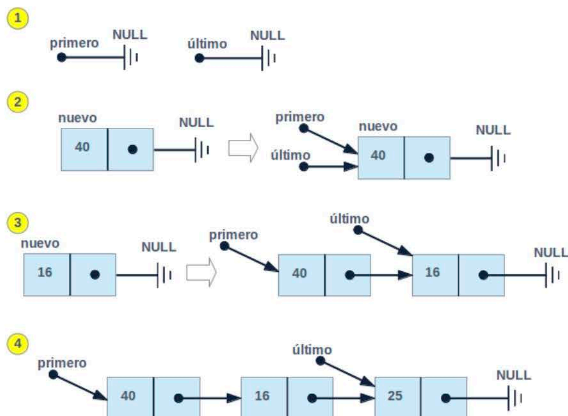

Figura 2.11. Proceso de inserción de elementos en una cola

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

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

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

Veamos la implementación de una cola en **Java**:

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

private Nodo primero;
private Nodo ultimo;

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

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

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

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

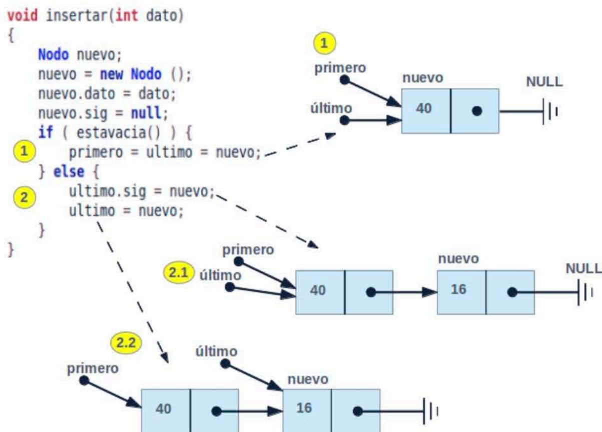

Figura 2.12. *Función insertar de la cola*

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

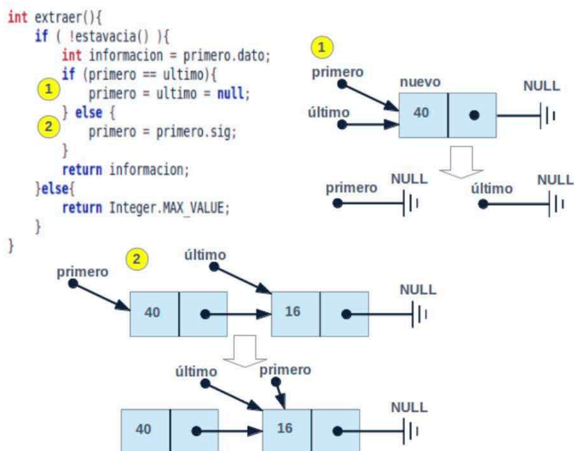

Figura 2.13. Función extraer de la cola

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

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

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

```
public class Cola {

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

```
    private Nodo primero;
    private Nodo ultimo;

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

    boolean estavacia(){
       …..
    }

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

    public int extraer(){
        …..
    }

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

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

2.3 TIPOS ABSTRACTOS DE DATOS

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

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

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

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

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

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?

a) Un puntero es una variable que apunta a una dirección de memoria donde reside un dato o una estructura de datos.

b) Las estructuras estáticas, al contrario que las dinámicas no utilizan punteros.

c) Las estructuras dinámicas nunca se utilizan cuando se sabe a ciencia cierta el número de elementos que van a contener.

d) El **método clone** permite duplicar un objeto.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Cuando no sabemos el número de elementos que va a tener la estructura, muchas veces es mejor resolver el problema utilizando una estructura dinámica.

b) En **Java** se pueden crear vectores o arrays de objetos.

c) El acceso a una matriz se haría de la siguiente forma: matriz[columna][fila].

d) El **método equals** permite discernir si dos referencias son el mismo objeto.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

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

b) Las sentencias byte[] temperaturas; y byte temperaturas[]; son lo mismo.

c) Una cadena de caracteres no es lo mismo que un vector o array de elementos de tipo **char**.

d) En **Java** cuando queremos agrupar una serie de datos relacionados entre sí, utilizamos una clase.

3 PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS

A la hora de programar no hay solamente que centrarse en escribir buen código sino que antes de ponerse manos a la obra hay que elegir las mejores herramientas como un buen **IDE** con buenas herramientas de depuración, un sistema de control de versiones y plantearse la posibilidad de escribir código para su posterior reutilización. A continuación se estudiarán en profundidad cada uno de estos conceptos.

3.1 EL ENTORNO DE DESARROLLO DE PROGRAMACIÓN

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 openjdk-7-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**.

Geany permite cambiar los comandos de compilación y de ejecución de **Java** no a nivel global, sino solamente para la aplicación. Quizás esta última opción es la mejor puesto que te va a permitir conocer con qué **JDK** está trabajando tu **IDE**.

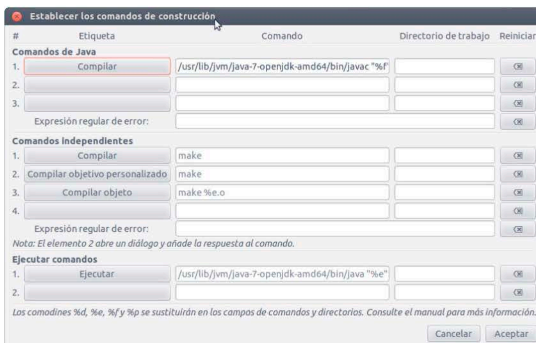

Figura 3.1. *Establecer comandos de construcción (menú construir)*

En mi caso (utilizo **Linux**), se puede ver en las imágenes cómo al instalar el **java-7-openjdk** lo que he hecho es que **Javac** (compilar) y **Java** (ejecutar), apunten al directorio donde está situado el **JDK** que he instalado en el equipo. En el caso que quieras cambiar de forma global en el sistema donde el sistema tiene que buscar **Java** en el equipo, deberás modificar el fichero **/etc/environment** (si es que utilizas **Linux**) añadiendo la siguiente línea:

```
JAVA_HOME="/usr/lib/jvm/java-7-openjdk-amd64"
```

En tu caso, dependiendo de la versión de **JDK** que hayas instalado tendrás que modificar el directorio java-7-openjdk-amd64 de la línea anterior.

Una vez configurado el **IDE** estamos en disposición de trabajar con nuestro entorno sin problemas.

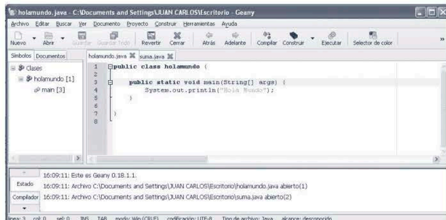

Figura 3.2 *IDE o Entorno de desarrollo Geany*

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 más 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 o profesionales.

RECUERDA

NetBeans y **Eclipse** son entornos de desarrollo libres.

3.1.1 ¿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 los comandos de construcción en **Geany** (recuerda que están en el menú construir).
- Prueba a compilar dentro y fuera del **IDE** el programa holamundo y comprueba que funciona ejecutándolo.

3.2 HERRAMIENTAS DE DEPURACIÓN

Prácticamente la mayoría de **IDE** tienen opciones de depuración que van a facilitar poner nuestros programas a punto. La depuración del código consiste en ir ejecutando paso a paso el código y analizar el contenido de las variables para comprobar que nuestro programa hace lo que debería o para solucionar algún fallo o error que nuestro programa esté cometiendo.

Cuando se va a depurar un programa lo normal es no comprobar paso a paso todo el programa sino el código del que tengamos dudas o que queramos revisar en profundidad. Para ello los **IDE** tienen unas herramientas que se llaman *breakpoint* o puntos en los que el programa se pausa y permiten ejecutar instrucción a instrucción.

Generalmente cuando se establece un punto de ruptura en el código, el **IDE** lo muestra con algún tipo de marca para que el programador sepa que a partir de ese punto puede trazar el programa y examinar las variables y las instrucciones.

Las herramientas paso a paso que tienen los **IDE** son de los siguientes tipos:

- **Step into**: se ejecuta la línea actual y en el caso que sea un procedimiento, método o función la depuración continúa dentro del mismo.
- **Step over**: se ejecuta la línea actual y en el caso que sea un procedimiento, método o función la depuración no continuará dentro del mismo.
- **Step return**: se siguen ejecutando las líneas del programa hasta que se encuentre un *return* y en ese caso parará para seguir depurando.

3.3 LA REUTILIZACIÓN DEL SOFTWARE

El concepto de reutilización surgió hace mucho tiempo debido a la demanda de más *software* y a precios más competitivos. En esa época todo el mundo reutilizaba *software* pero de una forma oportunista. Es decir, no se diseñaban las aplicaciones ni el *software* con la vista puesta en la reutilización del código, simplemente se reutilizaban aquellas partes que los desarrolladores sabían que le podía venir bien al nuevo programa o la nueva aplicación.

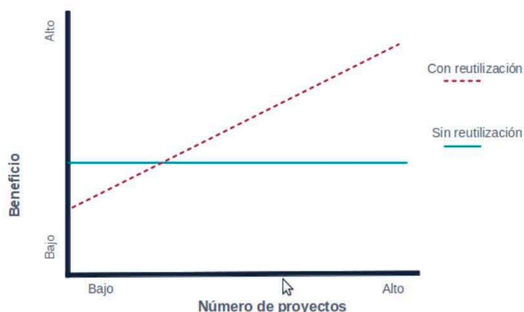

Figura 3.3. *Beneficio económico con la reutilización*

Existe una máxima con respecto a la reutilización y es que a más modularidad de los programas, más posibilidades de reutilización tiene dicho *software*. Entendemos **modularidad** en la creación de librerías o estructuras que permitan reutilizar la información.

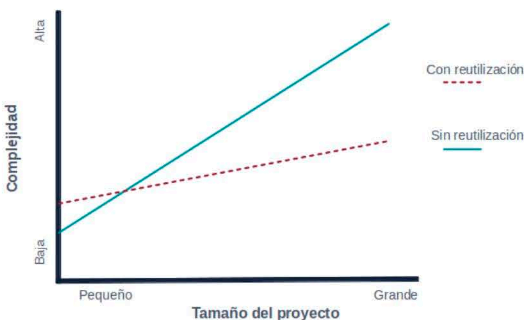

Figura 3.4. *Disminución de la complejidad con la reutilización*

En cuanto a la complejidad del *software*, como podemos ver en la figura anterior, con la reutilización disminuye. Los proyectos aunque sean más grandes no van a ser tan complejos y las personas menos expertas van a poder producir más gracias a la reutilización.

Los nuevos lenguajes de programación ya incorporan la reutilización. De hecho, es una de las características principales de los lenguajes de programación orientados a objetos. Con los componentes reutilizables es muy sencillo utilizar librerías de terceras personas. La ventaja de esto es que no tenemos que conocer los detalles de su implementación interna sino solamente su interfaz.

Uno de los inconvenientes de la reutilización es que hay que desarrollar el *software* pensando en ella, lo que hace que al principio tenga un coste.

RECUERDA

Una cosa es la reutilización puntual y otra diseñar y desarrollar para su posterior reutilización.

3.4 HERRAMIENTAS DE CONTROL DE VERSIONES

Cualquier programador con experiencia o en cualquier proyecto medianamente serio hay que tener un **control de versiones**. Una herramienta de control de versiones es imprescindible porque los programadores generalmente hacemos cambios a un *software*, luego nos retractamos y rehacemos los cambios, consultamos versiones anteriores, etc. Siempre que utilicemos información que va cambiando a lo largo del tiempo deberemos utilizar una herramienta de control de versiones.

Un sistema de control de versiones permite llevar un control de los distintos cambios que puede sufrir un producto a lo largo de su desarrollo o vida útil. Generalmente cuando en la elaboración de un producto intervienen muchas personas es imprescindible que un *software* haga el control de versiones puesto que de forma manual resultaría muy costoso y se podrían cometer numerosos errores.

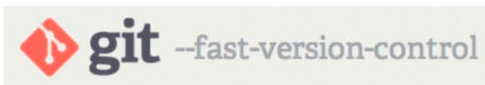

Figura 3.5. *Herramienta de control de versiones git*

3.4.1 ¿CÓMO SE ALMACENAN LAS VERSIONES?

Dependiendo de cómo se almacena el código, los sistemas de control de versiones se pueden clasificar en:

- **Centralizados**: los sistemas centralizados son más sencillos de gestionar. Existe un repositorio el cual centraliza el almacenamiento de todo el código. En el caso de que se deseen crear distintas versiones o ramas un responsable deberá aprobarlo. Algunas ventajas de los sistemas centralizados son el mayor control ejercido sobre el proyecto y la existencia de un número de versión.

- **Distribuidos**: los sistemas distribuidos mantienen un clon del repositorio por cada persona que lo utiliza. Eso quiere decir que cada usuario tiene un *backup* completo de la información que existe en el servidor. En caso de corrupción o fallo existen múltiples *backups* de los cuales se puede recuperar la información. Bastaría con que un usuario subiera la copia y se solucionaría el problema. No obstante, lo más común es que exista un repositorio principal que sirva de referencia que es el almacenado en el servidor. Algunas ventajas de los sistemas distribuidos son el almacenamiento de versiones inestables, en proceso o versiones de prueba las cuales pasarán a formar parte del repositorio común solamente en el caso que el usuario lo decida. Esta flexibilidad permite crear nuevas ramas para integrar nuevas funcionalidades o realizar reestructuraciones del código. También tiene la ventaja de poder trabajar de forma más autónoma dado que cada usuario trabaja con la copia de repositorio local, esto hace que el trabajo con sistemas distribuidos sea más ágil. También al trabajar los usuarios de forma distribuida se descarga más el servidor central recayendo gran parte de la potencia de cálculo en los distintos usuarios.

3.4.2 ¿CÓMO SE COLABORA EN UN SISTEMA DE CONTROL DE VERSIONES?

Existen dos formas de colaborar en un sistema de control de versiones, una es trabajar con el sistema de forma **exclusiva**. Este caso se utiliza mucho cuando cada usuario es responsable del desarrollo de algún elemento en concreto. Cuando el usuario está desarrollando ese elemento, lo que hace es comunicar al repositorio que está modificándolo y éste se encarga de bloquear las posibles modificaciones por parte de terceros. Una vez que lo tiene terminado, actualiza el repositorio y el elemento se desbloquea.

Otra manera que es muy utilizada en los proyectos de desarrollo de *software* libre es la forma de trabajo **colaborativa**. En este caso, cada usuario trabaja con su cuenta en su copia local y cuando termina los cambios que estaba realizando los sube al repositorio común. El problema que puede surgir con esta forma de trabajo es la aparición de conflictos al no estar coordinadas las distintas modificaciones.

3.4.3 ¿CÓMO SE TRABAJA EN UN SISTEMA DE CONTROL DE VERSIONES?

La forma en la que se trabaja con un sistema de control de versiones se llama *workflow* o **flujo de trabajo**. Existen diferentes formas de trabajar con un **SCV** y por lo tanto diferentes tipos de flujo de trabajo. A continuación se comentarán los más frecuentes:

Figura 3.6. *SCV con flujo de trabajo centralizado*

- **Flujo de trabajo centralizado o workflow centralizado**: es un sistema muy propio de la vieja escuela. Se suele utilizar un servidor centralizado y los desarrolladores van publicando sus actualizaciones de código en él. Estos sistemas funcionan de forma exclusiva, lo que implica que si alguien está modificando un elemento ningún otro desarrollador puede realizar actualizaciones del mismo.

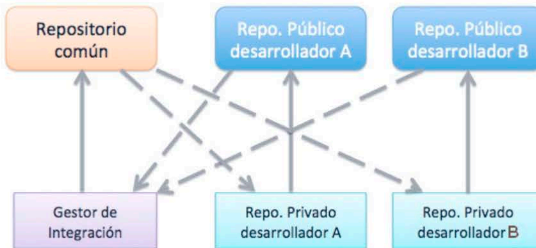

Figura 3.7. *Workflow con gestor de integración*

- **Flujo de trabajo con gestor de integración**: en ocasiones, para un mayor control sobre las versiones del proyecto se utiliza un gestor de integración que suele ser una persona del equipo de desarrollo que actualiza los trabajos en el repositorio común. Algo parecido a un guardia de tráfico. Los desarrolladores actualizan sus repositorios públicos y es el gestor de integración el encargado de actualizar el repositorio común con los trabajos públicos independientes de los desarrolladores. Este sistema suele ser muy utilizado en **SCV** distribuidos.

Figura 3.8. *Workflow con tenientes y dictador*

- **Flujo de trabajo con dictador y tenientes**: este tipo de flujo de trabajo se utiliza solamente en casos de proyectos masivos (como por ejemplo la evolución de un *kernel* de **Linux**). Es una evolución del modelo anterior. En proyectos muy grandes existen varias personas supervisoras de varias partes del proyecto llamadas tenientes y una persona llamada dictador que puede actualizar los cambios que sus tenientes le envían. En este sistema todos los desarrolladores tienen acceso al repositorio para poder clonarlo y crearse su propia copia del mismo.

3.4.4 SISTEMAS DE CONTROL DE VERSIONES CENTRALIZADOS: EL REPOSITORIO

Los sistemas de control de versiones centralizados se basan en el **repositorio centralizado**. Un **repositorio** es como un servidor de archivos con la salvedad que va recordando los cambios que hagamos a los ficheros y directorios que contiene. En estos repositorios se puede consultar versiones antiguas, examinar la historia de los ficheros y su información, etc. Este sistema es parecido al sistema de *backup Time Machine* de **Mac OS X**.

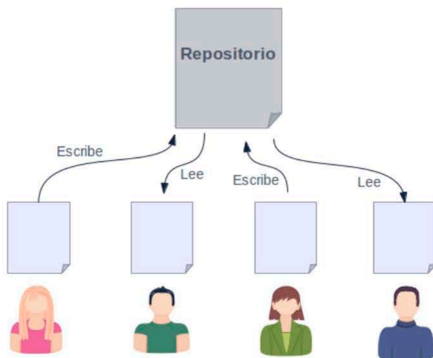

Figura 3.9. *Modo de trabajo con un repositorio*

Los repositorios deben de permitir acceder a múltiples usuarios de forma concurrente a la información (sino utilizaríamos un sistema de *backup* eficiente en vez de un repositorio). Además, los repositorios como están pensados para almacenar código, generalmente tienen herramientas para compilar y empaquetar proyectos, entender el código, etc.

Además, un repositorio debe ser capaz de responder a preguntas como ¿Quién fue la última persona en modificar este fichero? ¿Qué cambios ha hecho el usuario Emma a este fichero?¿Qué contenía este directorio el jueves pasado? Esas son las características por las que se usa un sistema de control de versiones y no un sistema de *backup* o servidor de ficheros.

3.4.5 SISTEMAS DE REPOSITORIO. MODELOS DE VERSIONADO

El problema que tiene todo el *software* de control de versiones es el acceso simultáneo a los mismos recursos por parte de los usuarios. Cuando varios usuarios acceden de forma simultánea a la misma información y la modifican, hay muchas probabilidades que la información se quede inconsistente y por esta razón dicho *software* de control de versiones no sería operativo.

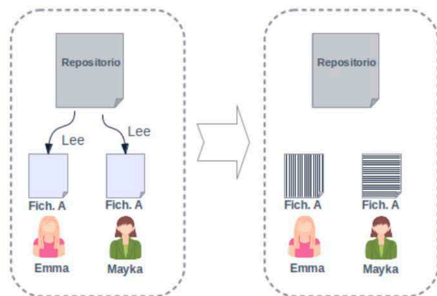

Figura 3.10. *Problemas del control de versiones*

Imaginemos el ejemplo de la figura anterior. Emma y Mayka leen del repositorio el mismo fichero (fichero **A**). Una vez modificados se disponen a sincronizarlo con el repositorio, ¿Qué pasaría?

Si Mayka graba su fichero y luego Emma graba el suyo se perderán las modificaciones hechas por Mayka.

En caso contrario, si es Emma la que graba primero se perderán sus modificaciones.

Por lo tanto, hay que establecer un mecanismo para que el problema anterior no suceda. Generalmente el *software* de control de versiones utiliza dos tipos de mecanismos de sincronización:

- La solución **bloqueo-modificación-bloqueo**: esta solución bloquea los ficheros cuando se van a modificar y una vez sincronizados con el repositorio y liberados se desbloquean. De esa manera el sistema está completamente seguro que no van a suceder inconsistencias en la información. El inconveniente de esta solución es que mientras que un fichero está siendo utilizado no es posible que los demás usuarios puedan trabajar con él.
- La solución **copia-modificación-fusión**: esta solución es más costosa de implementar pero más eficiente puesto que cada usuario puede hacerse su copia personal del fichero y trabajar con ella. Varios usuarios pueden trabajar en sus propias copias en paralelo y cuando sincronicen sus copias con el sistema de control de versiones se fusionarán en una versión final. Aunque el sistema de control de versiones ayuda a la fusión, siempre tiene que ser una persona la que valide el trabajo final.

Esta última solución es la utilizada por **Subversion** que es un *software* de control de versiones **Java** muy utilizado y que vamos a analizar en el apartado siguiente.

3.4.6 APACHE SUBVERSION

SUBVERSION®

Figura 3.11. Logo de Subversion

Subversion es un *software* libre de control de versiones de **Apache** que cuenta con las siguientes características:

- **Versionado de directorios**: subversion no solamente versiona ficheros sino directorios y árboles de directorios.
- Se puede trazar la historia de un fichero y además se pueden copiar y renombrar ficheros **eliminando su historia anterior** como si fuesen creados de nuevo.
- **Modificaciones atómicas**: se pueden hacer modificaciones solamente a trozos de código, de esa manera se puede volver atrás solamente con la parte del código que se quiera y no con todo el fichero.
- **Versionado de metadatos**: los ficheros tienen una serie de propiedades asociadas a él. Dichas propiedades también pueden ser versionadas.
- **Integración con Apache**: subversion se puede utilizar también dentro de **Apache** como una extensión lo cual hace que se puedan utilizar sus características como **SSH**, compresión, autenticación, autorización, etc.
- **Subversion** utiliza un **algoritmo de diferenciación** que funciona en binario, luego se pueden almacenar todo tipo de ficheros comprimidos en el repositorio y **Subversion** conocerá cualquier modificación de los mismos.
- **Mecanismo** *hard-link*: **Subversion** utiliza un sistema parecido al de los sistemas operativos para realizar copias de proyectos. De esa manera las copias son muy eficientes y rápidas.
- **Fácil de mantener y utilizable por otras aplicaciones**: gracias a que ha sido desarrollado en **C** con librerías y **APIs** perfectamente documentadas.

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un sistema de control de versiones permite llevar un control de los distintos cambios que puede sufrir un producto a lo largo de su desarrollo o vida útil.

b) La forma en la que se trabaja con un sistema de control de versiones se llama *workflow*.

c) Dependiendo de cómo se almacena el código, los sistemas de control de versiones se pueden clasificar en centralizados y distribuidos.

d) Existen dos formas de colaborar en un sistema de control de versiones, de forma exclusiva y de forma distribuida.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El flujo de trabajo con dictador y tenientes se utiliza solamente en casos de proyectos masivos.

b) Un repositorio es como un servidor de archivos con la salvedad que va recordando los cambios que hagamos a los ficheros y directorios que contiene.

c) Algunas ventajas de los sistemas distribuidos son el almacenamiento de versiones inestables.

d) La solución copia-modificación-fusión en un **SCV** es menos costosa de implementar pero más eficiente.

4 INTERFACES Y ENTORNOS GRÁFICOS

En este capítulo se aprenderá a crear interfaces de usuario con la librería **Swing** de **Java**. Un buen interfaz es fundamental a la hora de realizar una aplicación, por lo que en este tema también se explican conceptos sobre los mismos y se dan consejos para diseñar interfaces con éxito.

4.1 CARACTERÍSTICAS DE LAS INTERFACES, INTERACCIÓN HOMBRE-MÁQUINA

Los interfaces de usuario nacieron con el uso de los monitores en la informática. Anteriormente la interacción con el usuario se ceñía al intercambio de tarjetas perforadas entre usuario y máquina. Los primeros interfaces en monitores **CRT** eran textuales. Los monitores trabajaban con una pantalla que tenía 80 columnas por 25 líneas y muchas opciones de los interfaces se utilizaban con las famosas teclas de función (**F1**, **F2**,..., **F12**). Aunque parezca mentira los programadores hacíamos ventanas en modo texto y generalmente los interfaces eran sobrios pero eficientes.

Con la irrupción del ratón y los entornos gráficos los programas mejoraron en apariencia y en prestaciones puesto que un entorno gráfico podía incluir más información que un entorno en modo texto. La revolución llegó con los nuevos sistemas operativos en modo gráfico como **Windows** en los que se podían desarrollar aplicaciones con ventanas. Surgieron así los lenguajes **RAD** (*Rapid Application Development*) como *Visual Basic*, *Delphi*, etc.

Actualmente los programas tienen más opciones, son más complejos y diseñar el interfaz es una tarea a la que hay que dedicarle tiempo puesto que del éxito del interfaz depende también el éxito de la propia aplicación. También hay que tener en cuenta el sistema donde va a ejecutarse la aplicación. Actualmente las aplicaciones se desarrollan para ejecutarse en un navegador, como una **App** para ejecutarse en un dispositivo móvil o bien como aplicación tradicional.

Algunas de las características que se desean actualmente de todo interfaz serían las siguientes:

- **Claro**: Que la forma de interactuar con el interfaz sea sencilla y siga patrones lógicos (iconos, mensajes, etc.). Hay que evitar el mostrar información innecesaria. Ejemplos concretos serían el menú principal y la botonera de Google Drive que están bastante bien conseguidos, así como los sistemas de pestañas de Microsoft Office.

Figura 4.1 *Menú de los documentos de Google Drive*

- **Familiar o intuitivo**: ambos términos son sinónimos cuando hablamos de interfaces. El usuario tiene que saber interpretar de forma natural la información y características que el interfaz presenta.
- **Rápido**: más que deseable muchas veces es imprescindible. Limitar el número de accesos a la base de datos en ocasiones agiliza mucho el interfaz.
- **Configurable**: muchas veces la configuración a gusto del usuario hace ganar muchos adeptos a la aplicación.
- **Consistente**: por ejemplo en las suites ofimáticas se aprecia mucho que los interfaces no varíen entre versiones y que dentro de la misma versión, en los distintos programas de la suite sean lo más idéntico posible.

- **Atractivo**: por ejemplo, Google tiene un estilo minimalista que gusta a muchos usuarios.
- **Que ayude al usuario**: todos cometemos errores y muchas veces el poder rehacer operaciones o volver a un estado anterior, que guarde versiones, etc., son cosas que el usuario agradece.

4.2 DISEÑO DE INTERFACES

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**.

4.3 INTERFACES GRÁFICAS DE USUARIO. CREACIÓN DE NUESTRA PRIMERA APLICACIÓN CON SWING

La primera aplicación que se realiza con cualquier lenguaje de programación es el famoso **Hola Mundo**. Vamos a mostrar cómo 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.

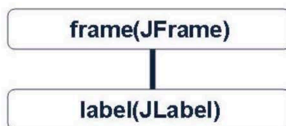

Figura 4.2. *Clases JFrame y JLabel*

```
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**.

4.3.1 LOS COMPONENTES SWING. LIBRERÍAS

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:

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.

Tabla 4.1. *Los componentes Swing*

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

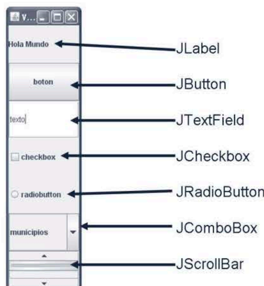

Figura 4.3. *Elementos de un interfaz en Java*

4.3.2 LOS CONTENEDORES SWING

Como antes se ha explicado, para realizar una aplicación **Java** necesitaremos utilizar un contenedor de nivel superior como pueden ser un **JFram**e, 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.

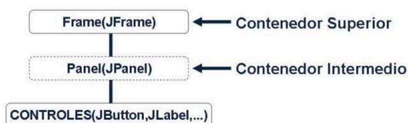

Figura 4.4. *Estructura de contenedores swing*

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**).

4.3.3 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 componentes 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 4.5. *Orientaciones de los componentes en un interfaz en Java*

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);
```

4.3.4 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.

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.

Tabla 4.2. *El método getSystemLookAndFeelClassName()*

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();}
```

4.4 PROGRAMACIÓN POR EVENTOS. CONCEPTO DE EVENTOS Y CONTROLADOR 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.

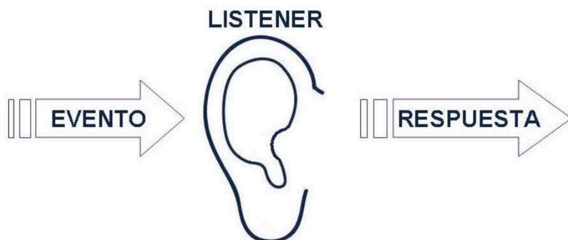

Figura 4.6. *Funcionamiento de un Listener*

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:

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. ...	1. Mover la barra de desplazamiento. 2. ...
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. ...	1. 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.

Tabla 4.3. *Listeners asociados a componentes*

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 4.7. Ejecución de un método tras el suceso 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 4.8. *Aspecto de la aplicación*

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).

4.5 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:

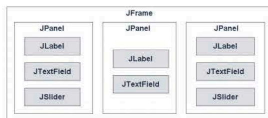

Figura 4.9. *Componentes de la aplicación de conversión*

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

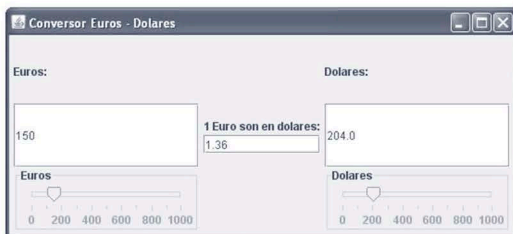

Figura 4.10. *Aspecto visual de la aplicación de conversión*

ACTIVIDADES

- 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 ejercicio.

  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();
    }
}
```

4.6 TÉCNICAS DE USABILIDAD

Todos tenemos claro qué es el concepto de usabilidad. Quizás no lo asociaremos al término usabilidad en primera instancia, pero si decimos que **usabilidad** es la facilidad que podamos tener en utilizar un programa o sistema, enseguida tendremos claro el concepto.

Todos nosotros hemos dicho alguna vez, "*este software es muy intuitivo*" o "*este sistema es fácil de usar*". Pues bien, la usabilidad aunque parezca complejo se puede medir en menor o mayor medida. Podemos medir, por ejemplo, en un *software* la facilidad con la que el usuario puede aprender a manejarlo, o también se podría medir la eficiencia del mismo (hay distintos procesadores de textos en los que para realizar una tarea determinada se necesita más tiempo que en otro por ejemplo). También podríamos medir la satisfacción del usuario con el sistema o la facilidad que tiene el usuario una vez dejado tiempo sin utilizar el sistema de volverlo a usar de una manera eficiente.

Muchas de esas características las vivimos los usuarios reales, por ejemplo, cuando nos cambian el aspecto de un programa o sistema operativo. En ocasiones acogemos la nueva versión de forma entusiasta y otras veces volvemos a la antigua por facilidad y eficiencia (**Windows 8.1** por ejemplo incorporó el botón de inicio que desapareció en **Windows 8**).

Ya que tenemos claro que la usabilidad es muy importante, hay que decir que más importante aún es la eficiencia. Por ejemplo, en el caso de los coches, todos sabemos que los automáticos son más sencillos de utilizar pero al final la gente utiliza coches de marchas porque aunque son más complejos al principio, son más divertidos y más versátiles.

La accesibilidad
La accesibilidad es aquella virtud o cualidad de un software que permite ser utilizado por el mayor número de personas posible incluyendo individuos con limitaciones o discapacidades.

4.6.1 LA SIMPLICIDAD COMO BANDERA DE LA USABILIDAD

En el diseño de interfaces, ya sean web, App o de aplicaciones tradicionales, la simplicidad es siempre un objetivo que hay que tener siempre presente durante el diseño. Simplicidad no quiere decir prescindir de elementos en una interfaz sino una correcta ordenación tanto en número como en forma. La simplicidad tiene que conducir a la eficiencia.

4.6.2 ALGUNOS CONSEJOS A LA HORA DE DISEÑAR UN INTERFAZ. RENDIMIENTO DEL INTERFAZ

A continuación se ofrecen algunos consejos que hay que tener en cuenta cuando se diseña un interfaz. Algunos parecen obvios pero muchas veces a la hora del diseño, aunque parezca increíble no se tienen en cuenta:

- En una interfaz hay que tratar de combinar textos, iconos e imágenes para hacer la aplicación más intuitiva y más cómoda. En la siguiente imagen se puede ver la combinación de texto e imágenes para presentar distinta información.

Figura 4.11. *Interfaz web de www.myfpschool.com*

- A la hora de diseñar un interfaz primero tenemos que analizar la funcionalidad y luego intentar dar con el diseño que explote al máximo dicha funcionalidad.
- En el momento de diseño, deberemos agrupar características bajo categorías comunes. De esa manera le será mucho más fácil al usuario encontrar cualquier opción.
- Hay que ponderar la cantidad de información que se presenta en cada interfaz y cuidar la navegación entre distintas páginas o ventanas.
- Por regla general, los menús de navegación o menús de la aplicación hay que aislarlos del resto del interfaz.
- Organizar los elementos del interfaz según una jerarquía visual. Hay que tener en cuenta que lo más importante tiene que estar más resaltado.
- El usuario tiene que reconocer las opciones. Es importante que el usuario cuando trabaja con un interfaz tiene que poder distinguir y reconocer las distintas opciones que se le presentan. Las opciones deben ser lo más intuitivas posibles.
- Control de errores en la interfaz. Todas las interfaces tienen que tener un control de errores eficiente. Tienen que avisar de forma inteligible al usuario del error o problema que ha surgido. Luego, si es necesario pueden mostrar un código que explique más en profundidad dicho problema.
- El sistema tiene que ayudar al usuario a no equivocarse. Por ejemplo, en la siguiente figura se puede ver cómo es más sencillo para el usuario elegir la fecha en un calendario que introducirla él mismo en el campo de texto. En el caso que sea el propio usuario el que tenga que escribir la fecha seguramente surgirán muchos problemas por errores tipográficos.

Figura 4.12. Detalle de un interfaz

- El interfaz tiene que ser lo más eficiente posible. En los interfaces hay que intentar adelantarse al usuario. Por ejemplo, en muchos interfaces de teléfonos móviles de sistemas operativos **IOS** y **Android** al seleccionar una foto, el mismo interfaz se adelanta y muestra las opciones que generalmente suelen hacer los usuarios (compartir , enviar por email, editar, etc.).

Figura 4.13. Opciones al seleccionar una fotografía en Android

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

Los interfaces de usuario nacieron con el uso de los monitores en la informática.

a) GridLayout coloca los componentes en filas y columnas.

b) La ventaja de **Swing** frente a **AWT** es que los componentes utilizados por la librería gráfica de **Swing** están programados con código nativo.

c) Con **AWC** la ventaja es que las aplicaciones se parecen mucho al *kit* de herramientas nativo subyacente.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un contenedor tiene predefinido por defecto un *Layout Manager* pero es posible e incluso deseable el cambiarlo cuando se realiza una aplicación.

b) La apariencia genérica de las ventanas de nuestras aplicaciones **Java** es algo que puede ser modificado según diferentes estilos.

c) A la hora de diseñar un interfaz primero tenemos que analizar el diseño y luego darle funcionalidad.

d) Un **Applet** es una aplicación **Java** que se ejecuta dentro de un navegador web en la máquina del cliente.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El **método pack** lo que hace es establecer el tamaño de un frame.

b) Los paneles son contenedores de componentes ligeros y estos a su vez pueden contener otros paneles.

c) La accesibilidad es aquella virtud o cualidad de un *software* que permite ser utilizado por el mayor número de personas posible incluyendo individuos con limitaciones o discapacidades.

d) Al contrario que los componentes **AWT** que son clases, los componentes **Swing** son una librería bastante completa.

5 ACCESO A BASES DE DATOS Y OTRAS ESTRUCTURAS

Prácticamente la mayoría de los accesos a bases de datos de cualquier aplicación son sobre **bases de datos relacionales**. Una base de datos relacional almacena datos en tablas de tal manera que esos datos puedan ser almacenados y recuperados de una forma eficiente. Las tablas se componen de una serie de objetos o filas (*rows* en inglés) las cuales tienen los mismos elementos.

Un **SGBD** (Sistema Gestor de Bases de Datos) o lo que es lo mismo **DBMS** (*Data Base 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**).

5.1 OBJETOS DE LA BASES DE DATOS. 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*).

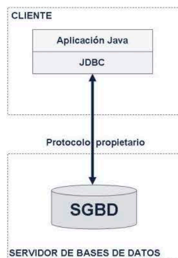

Figura 5.1. *Modelo de arquitectura basado en dos niveles*

En los modelos de arquitectura en dos niveles, el cliente accede directamente a los datos a través del driver **JDBC**.

Los comandos son enviados a la base de datos y los resultados son devueltos a la aplicación cliente. Es una configuración cliente/servidor donde la máquina del usuario que corre la aplicación **Java** es el cliente y el servidor es donde reside la base de datos. Servidor y cliente pueden estar en máquinas diferentes en la misma red local o a través de Internet.

Modelo de arquitectura basada en **tres niveles** (*three-tier*).

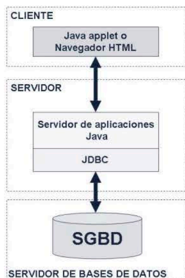

Figura 5.2. *Modelo de arquitectura basado en dos niveles*

En la arquitectura en tres niveles hay una capa intermedia donde reside la lógica del negocio. En esta capa intermedia o *middleware* está situado el servidor de aplicaciones el cual envía los comandos que recibe de la máquina cliente al gestor de bases de datos y los resultados se los vuelve a enviar al cliente. Esta capa intermedia generalmente suele aportar una mayor seguridad y un acceso más controlado a los datos ganando en muchas ocasiones una mejora del rendimiento. Cada vez más se está utilizando **Java** en la programación de la capa intermedia en detrimento de otros lenguajes de programación como **C** o **C++**.

JDBC puede ser implantado sin problemas en cualquiera de estos dos modelos anteriores.

5.1.1 ¿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.

¿SABÍAS QUE?

Yo en vez de instalar solo **MySQL** he decidido instalar **XAMPP** versión *Lite* (software libre gratuito) y de esa manera instalo **Apache**, **MySQL**, **PHP** y **PhpMyAdmin**. Eso me permite administrar y manejar la base de datos desde un navegador web.

5.2 CONEXIONES PARA EL ACCESO A DATOS

Antes de trabajar con la base de datos hay que establecer una conexión con la misma. **JDBC** se conecta a las bases de datos utilizando una de estas dos clases:

- **DriverManager**: es la forma más sencilla de conectarse a una base de datos. La conexión con la base de datos se realiza especificando una dirección **URL**.
- **DataSource**: esta clase hace que los detalles sobre la base de datos a la que se conecta sean transparentes a la aplicación.

En los ejemplos que se van a ver a continuación se va a utilizar la clase **DriverManager** para establecer conexiones con la base de datos.

RECUERDA

No olvides importar el paquete java.sql cuando tu programa utilice JDBC. Recuerda ejecutar la siguiente sentencia al inicio de tu programa .java:

```
import java.sql.*;
```

El siguiente código permite realizar una conexión con una base de datos **MySQL**:

```
try {
connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/
test","andrés","pelusilla");
System.out.println("Connection succeed!");
}
catch (Exception e) {
e.printStackTrace();
}
```

Nótese que se utiliza el **método getConnection** de la clase **DriverManager**. Para establecer la conexión se pasan tres parámetros a este método:

- La **URL**: "*jdbc:mysql://localhost:3306/test*", donde:
  - **localhost**: es la dirección de la máquina donde reside la base de datos.
  - **3306**: es el puerto donde escucha la base de datos.
  - **test**: es la base de datos a la que se conectará el programa.
- El **usuario** con el que se conecta a la base de datos: "andres".
- La **password** del anterior usuario: "pelusilla".

5.3 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 Method)

        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)

        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Sou
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:1464)
        at test.viewTable(test.java:26)
        at test.main(test.java:48)
SQLState: 42S22
Error Code: 1054
Message: Unknown column 'datos' in 'field list'
Presione una tecla para continuar . . .
```

Figura 5.3. *Excepción lanzada por el programa*

La figura anterior muestra el resultado del tratamiento de una excepción **Java**.

5.4 CREACIÓN Y CARGA DE DATOS EN TABLAS

Para los siguientes ejemplos se va a crear la siguiente estructura de tablas.

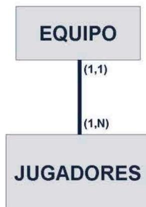

Figura 5.4. *Relación entre las tablas equipo y jugadores*

En esta estructura existen dos tablas, la de los equipos y la de los jugadores. Cada jugador tiene un equipo por el que juega y un equipo se compone de 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));
```

5.4.1 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.

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.

5.4.2 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();
      }
  }
```

5.5 RECUPERACIÓN DE LA INFORMACIÓN DE LA BASE DE DATOS

Para recuperar la información de la tabla equipos se puede utilizar el método que se describe a continuación:

```
public static void verEQUIPO(Connection con, String BDNombre) throws SQLException {
    Statement stmt = null;
    String query = "select EQ_NOMBRE ,ESTADIO ,POBLACION ,PROVINCIA "+
                              " from " + BDNombre + ".EQUIPO";
    try {
      stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String equipo = rs.getString("EQ_NOMBRE");
        System.out.println("Equipo: "+equipo);
        String estadio = rs.getString("ESTADIO");
        System.out.println("Equipo: "+estadio);
        String poblacion = rs.getString("POBLACION");
        System.out.println("Equipo: "+poblacion);
        String provincia = rs.getString("PROVINCIA");
        System.out.println("Equipo: "+provincia);
        System.out.println("**********************************************");
      }
    } catch (SQLException e ) {
          printSQLException(e);
    } finally {
      stmt.close();
    }
}
```

En el método anterior se puede observar que se utiliza el objeto **ResultSet**, el cual representa un conjunto de datos recuperado de una base de datos (matriz de datos). En este procedimiento se crea un objeto **ResultSet** llamado **rs** 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*.

5.5.1 LA INTERFAZ RESULTSET

La interfaz *Resultset* como hemos visto, tiene métodos para recuperar y manipular los datos relativos a comandos **SQL** realizados a una base de datos. Existen distintos tipos de objetos *ResultSet* dependiendo de sus características.

Tipos de *ResultSet*:

- **TYPE_FORWARD_ONLY**: este cursor es el cursor por defecto. Los ejemplos anteriores están realizados con este cursor. Como su nombre indica es un cursor unidireccional y solo se mueve en un sentido hacia delante (desde la primera fila hasta la última).
- **TYPE_SCROLL_INSENSITIVE**: este cursor puede moverse hacia delante y hacia 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 el interface ResultSet.
- **CONCUR_UPDATABLE**: el objeto ResultSet NO 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.

5.5.2 OTRA MANERA DE RECUPERAR LOS DATOS DE UNA TABLA

Existe otra manera diferente de recuperar los datos de un Resultset, en vez de utilizar los nombres de los campos en los métodos **getString**, **getBoolean**, **getLong**, etc. La solución es llamar a los campos por el orden que ocupan en la sentencia **SQL**, comenzando por el número 1.

```
public static void verEQUIPO(Connection con, String BDNombre) throws SQLException {
    Statement stmt = null;
    String query = "select EQ_NOMBRE ,ESTADIO ,POBLACION ,PROVINCIA "+
                           " from " + BDNombre + ".EQUIPO";
    try {
      stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String equipo = rs.getString(1);
        System.out.println("Equipo: "+equipo);
        String estadio = rs.getString(2);
        System.out.println("Equipo: "+estadio);
        String poblacion = rs.getString(3);
        System.out.println("Equipo: "+poblacion);
        String provincia = rs.getString(4);
        System.out.println("Equipo: "+provincia);
        System.out.println("*********************************************");
      }
```

```
      } catch (SQLException e ) {
            printSQLException(e);
      } finally {
        stmt.close();
      }
  }
```

Este tipo de recuperación de información se utiliza cuando recuperamos varias columnas que tienen el mismo nombre. De la manera anterior no sería posible recuperar la información y de esta manera sí.

RECUERDA

El **método getString** puede servirnos para recuperar datos **CHAR** y **VARCHAR** de las bases de datos. También es posible recuperar datos numéricos de la base de datos pero hay que tener en cuenta que estos serán convertidos a *String*.

5.5.3 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 relativa a la actual posición.
- **absolute**(**int row**): coloca el cursor en la posición especificada en el parámetro *row*.

5.6 MODIFICACIÓN Y ACTUALIZACIÓN DE LA BASE DE DATOS

En este apartado veremos cómo se modifican y actualizan daton en la base de datos utilizando la sintaxis clásica y los objetos Resulset.

5.6.1 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();
    }
}
```

5.6.2 MODIFICACIÓN DE DATOS EN LAS TABLAS UTILIZANDO RESULTSET

Como se puede ver en el siguiente ejemplo, en **Java** se pueden crear **ResultSet bidireccionales** y actualizables. El siguiente método actualiza la edad de los jugadores y le suma el valor introducido en el parámetro entero **cuantoMas**.

```
  public static void modificaEdadJugadores(Connection con, String BDNombre,int cuantoMas)
throws SQLException {
    Statement stmt = null;
    try {
      stmt = con.createStatement();
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_
UPDATABLE);
      ResultSet rs = stmt.executeQuery(
        "SELECT * FROM " + BDNombre + ".JUGADORES");
```

```
        while (rs.next()) {
          int i = rs.getInt("EDAD");
          rs.updateInt("EDAD", i + cuantoMas);
          rs.updateRow();
        }

      } catch (SQLException e ) {
        printSQLException(e);
      } finally {
        stmt.close();
      }
    }
```

Como se estudió anteriormente, la propiedad **TYPE_SCROLL_SENSITIVE** hace que el objeto *ResultSet* creado pueda moverse bidireccionalmente de forma relativa a su posición actual 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.

5.6.3 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**. Al ejecutar el método **insertRow**() del objeto ResultSet se insertarán los datos tanto en el ResultSet como en la base de datos.

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Un **SGBD** es lo mismo que un **DBMS**.

b) La **API JDBC 4.0** está dividida en dos paquetes, java.sql y javax.sql.

c) Para trabajar con **JDBC** se necesitará crear un entorno mínimo que permita compilar y ejecutar los programas **Java**.

d) Los objetos *statement* no se cierran automáticamente, se recomienda cerrar los objetos statement mediante el comando **stmt.close**();.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Una **SQLException** puede haber sido lanzada debido a una o varias causas.

b) Una base de datos relacional almacena datos en tablas de tal manera que esos datos puedan ser almacenados y recuperados de una forma eficiente.

c) Las tablas se componen de una serie de objetos o filas (*rows* en inglés) las cuales tienen los mismos o distintos elementos.

d) Un **SGBDR** es lo mismo que un **RDBMS**.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El puente *software* de **Java JDBC** proporciona acceso a gestores de bases de datos a través de **ODBC**.

b) En los modelos de arquitectura en tres niveles, el cliente accede directamente a los datos a través del driver **JDBC**.

c) La clase **JDBC** DriverManager define los objetos desde los que se pueden conectar las aplicaciones **Java** a un controlador **JDBC**.

d) En la **URL** *jdbc:mysql://localhost:3306/test*, localhost es la dirección de la máquina donde reside la base de datos.

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) En la **URL** *jdbc:mysql://localhost:3306/test*, localhost es la base de datos a la que se conectará el programa.

b) En la arquitectura en tres niveles hay una capa intermedia donde reside la lógica del negocio.

c) **PreparedStatement** se utiliza para ejecutar comandos **SQL** con o sin parámetros de entrada ya precompilados.

d) La **API JDBC** ofrece un acceso a bases de datos relacionales desde **Java**.

6 PRUEBAS

Los proyectos de desarrollo de *software* han sufrido tradicionalmente problemas de calidad, tanto en el propio proceso de desarrollo como en los productos que entregan. Estos problemas surgen habitualmente en las desviaciones de plazos y esfuerzo sobre los valores previstos y en la aparición de fallos durante la implantación y mantenimiento de dichos productos.

Las **pruebas de software** o *testing* son aquel conjunto de procesos que 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).

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).

6.1 OBJETIVOS DE LAS PRUEBAS

El principal objetivo de las pruebas de un sistema o un *software* es sacar a relucir el mayor número de defectos, fallos o errores existentes. Existen un gran número de razones por la 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, 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.

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 que proporcionan 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.

6.2 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 este 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 de interfaces, prueba funcional, prueba de sistema, prueba de fin a fin, prueba de sanidad, prueba de regresión, prueba de aceptación, prueba de carga, prueba de estrés, prueba de performance, prueba de instalación y desinstalación, prueba de recuperación, prueba de seguridad, prueba de compatibilidad, prueba de exploración, prueba de anuncio, prueba de comparación, prueba alfa, prueba beta, prueba de mutación.

Pasemos a describir en profundidad algunas de las pruebas más conocidas:

- **Prueba de caja negra y caja blanca**: en este tipo de pruebas solo se tienen en cuenta las entradas y salidas de la aplicación o sistema. Lo que se va a testear en este tipo de pruebas es **qué** es lo que hace el sistema. Por el contrario, las pruebas de caja blanca prueban el sistema pero atendiendo a **cómo** lo hace. En las pruebas de caja blanca al contrario que las de caja negra sí es importante los aspectos internos del *software*.
- **Prueba de estrés**: este tipo de pruebas se realiza sobre el *software* ya acabado o en fase alfa. Es una de las típicas pruebas que se realiza una vez el *software* se acerca a su estado final y el objetivo del *testing* es generar una gran cantidad de datos para verificar si el *software* es robusto y no se ve afectado por la concurrencia. Muchas veces cuando se desarrolla un *software* se prueba con una serie de casos de prueba a menudo muy sencillos y de forma aislada por el programador o analista. En este caso se ponen a trabajar con el sistema múltiples equipos como cliente y generalmente se utiliza *software* que imita el uso que le daría un usuario concreto dando altas, bajas modificaciones, accediendo a datos, etc. Esto provoca que el sistema se enfrente a una situación más real que hasta ahora. De esta prueba se tomarán tiempos y se podrá conocer cuál será el comportamiento del sistema una vez implantado.
- **Pruebas de integración**: generalmente los proyectos están desarrollados por varios programadores o varios equipos de programación los cuales trabajan sobre una parte del mismo. Existe un momento crítico y es cuando dichos programadores o grupos integran su trabajo para compilar un proyecto completo. Muchas veces en ese momento surgen errores, incompatibilidades o fallos debidos a la integración de los distintas partes del *software*.
- **Pruebas de interfaces**: en algunos proyectos se desarrollan varios módulos de una aplicación concreta o varias aplicaciones que comparten información entre sí. Este tipo de pruebas tratan de verificar que las aplicaciones o módulos son capaces de comunicarse entre sí de una manera eficiente y efectiva.

Una práctica popular y cada vez más habitual es la de distribuir de forma gratuita una versión no final del producto para que sean los propios consumidores los que la prueben. En ambos casos a la versión del producto en pruebas anterior a la **versión final 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*), dónde se comprueba cada funcionalidad del programa completo en entornos de producción.

Haciendo historia...

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

6.3 PLANIFICACIÓN DE LAS PRUEBAS

Las pruebas 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 finaliza 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. En el siguiente apartado se estudiará el plan de pruebas y el proceso de pruebas con más profundidad.

¿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.

6.4 PROCESO DE PRUEBAS Y DOCUMENTACIÓN DE LAS MISMAS

El proceso de pruebas consta de una serie de pasos que se van a abordar uno detrás de otro. Cada uno de estos pasos estará debidamente documentado. Pasemos a estudiar en profundidad cada uno de ellos:

6.4.1 PLANIFICACIÓN DE LAS PRUEBAS: EL PLAN DE PRUEBAS

En este primer paso se definen los objetivos de la prueba, en definitiva **qué** se desea probar. Todo el mundo pensará que el objetivo será detectar errores, pero muchas veces eso no es lo más importante. En ocasiones se busca que el sistema ofrezca un rendimiento determinado, otras que el interfaz cumpla unas determinadas características, etc. En ocasiones, el *software* no presenta errores pero no cumple ciertos requisitos y no pasa las pruebas.

En este momento se debe decidir quién hace la prueba y bajo qué condiciones se va a realizar. Hay que tener en cuenta que los propios programadores no son los más adecuados para realizar las pruebas puesto que se limitarán a realizar las pruebas que ya han ido haciendo durante el desarrollo del *software* y por lo tanto el objetivo de la prueba no tiene mucho sentido.

También hay que describir la estrategia que se va a seguir, es decir, cómo se va a realizar la prueba. Es importante en este momento decidir los recursos que se le va a asignar a las pruebas incluyendo personas, las máquinas donde se va a realizar, el tiempo que se va a dedicar etc.

Hay que tener en cuenta que es imposible probar todo, la prueba exhaustiva no existe. Muchos errores del sistema saldrán en producción cuando el *software* ya está implantado pero se intentará siempre que sea el mínimo número de ellos.

En esta fase se elaborará un plan de pruebas que debería tener al menos cubiertos los siguientes apartados:

- **Introducción**: breve introducción del sistema describiendo objetivos, estrategia, etc.
- **Módulos o partes del software a probar**: detallar cada uno de estas partes o módulos.
- **Características del software a probar**: características individuales o conjuntos de ellas.
- **Características del software a no probar**.
- **Enfoque de las pruebas**: en el que se detallan entre otros las personas responsables, la planificación, la duración, etc.
- **Criterios de validez o invalidez del software**: en este apartado se registra cuando el *software* puede darse como válido o como inválido especificando claramente los criterios.
- **Proceso de pruebas**: se especificará el proceso y los procedimientos de las pruebas a ejecutar.
- **Requerimientos del entorno**: incluyendo niveles de seguridad, comunicaciones, necesidades *hardware* y *software*, herramientas, etc.
- **Homologación o aprobación del plan**: este plan deberá estar firmado por los interesados o responsables del mismo.

Las demás fases del proceso de pruebas como se puede entender son el mero desarrollo del plan de pruebas anterior.

6.4.2 PREPARACIÓN DE LOS DATOS DE PRUEBA

En esta fase de las pruebas se diseñarán los casos de pruebas. Los casos de prueba se diseñan para que haya una alta probabilidad de encontrar un fallo. Es importante en esta fase no ser redundante. Si se ha probado algo y funciona no hace falta probarlo más.

Tenemos que tener en cuenta que la prueba no debe ser muy sencilla ni muy compleja. Si es muy sencilla no va a aportar nada y si es muy compleja quizás sea difícil encontrar el origen de los errores.

RECUERDA

Probar es ejecutar casos de prueba uno a uno pero el que un software pase todos los casos de prueba no quiere decir que el programa está exento de fallos.

Como hemos visto, las pruebas solo encuentran o tratan de encontrar aquellos errores que van buscando, luego es muy importante realizar un buen diseño de las pruebas con buenos casos de prueba puesto que se aumenta de esta manera la probabilidad de encontrar fallos.

RECUERDA

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y salidas esperadas diseñadas para un objetivo concreto.

6.4.3 CODIFICACIÓN DE LAS PRUEBAS

Una vez diseñados los casos de prueba hay que generar las condiciones necesarias para poder ejecutar dichos casos de prueba. Habrá que codificarlos en muchos casos generando *set* o **conjuntos de datos**. En estos *set* de datos hay que incluir tanto datos válidos, datos inválidos o incluso algunos datos fuera de rango o disparatados.

También habrá que preparar las máquinas sobre las que se van a hacer las pruebas instalando el *software* necesario, los usuarios de sistema, realizar carga del sistema, etc.

6.4.4 EJECUCIÓN DE LAS PRUEBAS

Una vez definidos los casos de prueba y establecido el entorno de las pruebas es el momento de la ejecución de las mismas.

Se irán ejecutando los casos de prueba uno a uno y en el momento que detectemos algún error, lo que hay que hacer es aislarlo y anotar la acción que se estaba probando, el caso, el módulo, la fecha, hora, los datos utilizados, etc. De esa manera se intentará documentar lo más detalladamente posible el error.

En el caso que se produzcan errores aleatorios también hay que registrarlos anotando este hecho.

RECUERDA

Casi siempre que se encuentra un error en un módulo de software, si se insiste se encontrarán más.

6.4.5 GENERACIÓN DEL INFORME FINAL DE LAS PRUEBAS

Esta es la fase final de las pruebas. En los informes se detallarán de manera formal las anotaciones realizadas durante las pruebas. En ellas se clasificarán los errores o fallos encontrados clasificando su naturaleza y su importancia. Generalmente estos informes van acompañados de un análisis de resultados y están firmados por las personas responsables de haber realizado el *testing*.

6.5 PRUEBAS DE RENDIMIENTO

El desarrollo de cualquier *software* tiene como único objetivo satisfacer una necesidad planteada por un cliente. Pero ¿cómo puede saber un desarrollador si el producto construido corresponde exactamente con lo que el cliente pidió? y ¿cómo puede un desarrollador estar seguro de que el producto que ha construido va a funcionar correctamente? Para esto es recomendable que el producto de *software* sea evaluado a medida que se va construyendo.

Por lo tanto, se hace necesario llevar a cabo, en paralelo al proceso de desarrollo, un proceso de evaluación o comprobación de los distintos productos o modelos que se van generando, en el que participarán desarrolladores y clientes, lo que dará como resultado un mejor producto, aumentando de manera significativa la satisfacción del usuario final.

El **origen** del interés actual por la calidad, se puede explicar recurriendo al estudio de la evolución en la comercialización de los productos. En un mercado tan competitivo como el de hoy día no basta con producir y distribuir masivamente los productos o servicios: vender es lo importante y sólo se produce con la seguridad de la aceptación del cliente.

La **calidad** del *software* se convierte así en un **objetivo fundamental** para las empresas junto a los dos parámetros clásicos de su gestión: dinero y tiempo. Lo que prima es la adaptación a las necesidades del cliente y esto lleva a investigar primero cuáles son esas necesidades (investigación de mercados) para luego definirlas en forma de requisitos a cumplir (especificaciones).

Las **características** específicas del *software*, convierten a la tarea de garantizar su calidad en algo mucho más difícil que por ejemplo el desarrollo de un producto a través de un proceso industrial.

La **calidad es pues un parámetro fundamental** que permite obtener una idea sobre su adecuación a los fines para los que ha sido construido, y más en el mundo actual en el que las aplicaciones informáticas se están introduciendo en todos los ámbitos, siendo pues imprescindible que esté libre de defectos y errores.

Los años noventa fueron escenario de una gran **crisis del software**, cuyas principales características fueron:

- Calidad insuficiente del producto final.
- Estimaciones de duración de proyectos y asignación de recursos inexactas, con el problema que ello conlleva.
- Escasez de personal cualificado en un mercado laboral de alta demanda.
- Tendencia al crecimiento del volumen y complejidad de los productos.

Con el tiempo se ha constatado que la calidad no se mide solamente por unos parámetros de funcionamiento sino que hay otros aspectos que son importantes como el soporte, es decir, el respaldo organizacional que tiene un producto como son la formación, asistencia a problemas inesperados y el mantenimiento permanente y efectivo.

Para evaluar dicha calidad se lleva a cabo la **evaluación y rendimiento de las aplicaciones**.

Las mediciones de rendimiento de un *software* pueden estar **orientadas hacia el usuario** (ej. tiempos de respuesta) u **orientadas hacia el sistema** (ej. uso de la **CPU**). Son medidas típicas del rendimiento diferentes variables de tiempo (tiempo de retorno, tiempo de respuesta, tiempo de reacción), la capacidad de ejecución, la carga de trabajo, la utilización, etc.

Para evaluar el *software* es necesario contar con criterios adecuados que nos permitan analizar el *software* desde diferentes puntos de vista.

Las **pruebas de rendimiento** son las que se realizan para determinar lo rápido que realiza una tarea un *software* en condiciones particulares de trabajo. Sirven también para evaluar otros atributos de la calidad del *software* como son la **escalabilidad**, **fiabilidad** y el **uso de los recursos**.

En ocasiones se emplean las **pruebas de carga** que observan el comportamiento de una aplicación bajo una cantidad de peticiones esperada como un número elevado de usuarios concurrentes usando la aplicación que realizan un número elevado de transacciones y muestra los tiempos de respuesta de las transacciones.

Y también son útiles las **pruebas de estrés** que van doblando el número de usuarios que manejan la aplicación hasta que se rompe, para determinar la solidez de la aplicación en momentos de carga extrema.

Existen otras pruebas como la **prueba de estabilidad** que somete la aplicación a una carga continuada, y las **pruebas de picos**, donde la carga va cambiando.

Un **benchmark** es una aplicación o conjunto de aplicaciones cuya finalidad es evaluar el rendimiento de un ordenador. Existen cuatro categorías generales de pruebas de comparación:

- **Pruebas de aplicaciones-base** que se encargan de ejecutar y cronometrar los tiempos de las mismas.
- **Pruebas playback** que usan llamadas al sistema durante actividades específicas de una aplicación como uso de disco o llamadas a rutinas de gráficos, ejecutándolas aisladamente.
- **Pruebas sintéticas** que enlazan actividades de la aplicación en subsistemas específicos.
- **Pruebas de inspección** que no intentan imitar la actividad sino que las ejecuta directamente en su entorno productivo.

Existen múltiples programas para llevar a cabo este tipo de pruebas como **Winstone** o **Winbench** de **ZDNet**.

6.6 NORMAS DE CALIDAD

Generalmente cuando creamos un programa debemos seguir unas normas de calidad o mejor dicho, de estilo de programación. A continuación se dará un resumen muy reducido de normas de estilo que generalmente se siguen cuando se programa en **Java**:

No tomes estas reglas al pie de la letra. Lo importante cuando se escribe un programa es que sea fácilmente entendible. Siguiendo estas reglas se pueden escribir programas complicados de entender. Utiliza el sentido común.

Nombres

- Las variables deben comenzar con minúsculas.

```
linea
```

- Las variables compuestas deben escribirse combinando mayúsculas y minúsculas.

```
lineaDocumento
```

- Las constantes siempre van en mayúsculas.

MAX_LINEAS

- Los nombres de procedimientos y métodos deben escribirse combinando mayúsculas y minúsculas.

```
calcularTotal()
```

- Los iteradores o variables que recorren una serie de valores deberían de nombrarse i, j, k, l, etc.

```
for (int i = 0; i < nTables; i++) {
:
}
```

Líneas muy largas

Una buena solución es dividir las líneas muy largas en partes para hacerlas más legibles en pantalla.

```
System.out.println("Esta es una línea muy larga que" +
                             " la he dividido en dos partes.");
```

Arrays: Colocar los *brackets* al lado del tipo.

```
double[] lista; // NO: double lista[];
```

- **Bucles**:
  - Utiliza siempre *while* en vez de *do while*. Los programas son más legibles siempre con while porque la condición está al principio y no al final. Además los *do-while* no son necesarios, siempre se puede cambiar un *while* por un *do-while*. Y por último, si reduces el número de estructuras los programas siempre serán más homogéneos y sencillos de entender.
  - Evitar el uso de *break* en bucles.

- **Sentencias condicionales**:

  Los *if* e *if -else* deberían tener este aspecto:

```
if (condiciones) {
sentencias;
}

if (condiciones) {
sentencias;
} else {
sentencias;
}
```

Uso de espacios en blanco

En muchas sentencias como por ejemplo con los operadores es más legible introducir espacios en blanco entre ellos:

a = (b + c) * d; // NO: a=(b+c)*d;

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las pruebas de *software* o *testing* son aquel conjunto de procesos que permiten verificar y validar la calidad de un producto *software*.

b) El *testing* puede probar la presencia de errores pero no la ausencia de ellos.

c) El *testing* sirve para corroborar la ausencia de errores.

d) En las pruebas de caja blanca son importantes los aspectos internos del *software*.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Los defectos de forma son aquellos en los que el programa no realiza lo que el usuario espera.

b) La prueba de verificación consiste en demostrar que un programa cumple con sus especificaciones.

c) La manera más útil de probar un *software* es en el mismo entorno de explotación.

d) Nunca se debe testear el *software* en un entorno de explotación.

3 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las pruebas de integración tratan de verificar que las aplicaciones o módulos son capaces de comunicarse entre sí de una manera eficiente y efectiva.

b) Los programadores a veces son uno de los principales factores en la generación de errores.

c) En la fase **RTM** *testing* se comprueba cada funcionalidad del programa completo en entornos de producción.

d) La prueba de validación se encarga de comprobar que el programa da la respuesta que espera el usuario.

4 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Las presiones de tiempo hacen que el número de errores aumente.

b) Si un *software* pasa todos los casos de prueba, podremos decir sin equivocarnos que el programa está exento de fallos.

c) Un tester debe tener una actitud de probar para romper.

d) Más del 80% de los errores cometidos provienen de las primeras fases del ciclo de vida.

7 HERRAMIENTAS DE GENERACIÓN DE PAQUETES

En este capítulo se verán las herramientas de generación de paquetes más usuales en **Java**. Generalmente se utilizan los ficheros **JAR** por ser un sistema **multiplataforma**. Un fichero **JAR** se puede crear desde el **IDE** o bien manualmente. Nosotros veremos esta última opción porque nos va a permitir entender mejor cómo funcionan.

Aparte de **JAR** veremos otras herramientas como *wrappers*, instaladores y **JWS** en el segundo apartado de este tema.

7.1 LOS FICHEROS JAR: FUNCIONES Y CARACTERÍSTICAS

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:

- **Crear** un fichero **JAR**.
- Ver el **contenido** del **JAR**.
- **Extraer** los ficheros de un **JAR**.
- **Ejecutar** la aplicación contenida en un **JAR**.

7.1.1 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**. Veamos cómo se ejecutaría este comando en **Windows**:

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

7.1.2 VER EL CONTENIDO DEL JAR

El contenido del **JAR** se muestra utilizando los parámetros **tf**. Veamos cómo se ejecutaría este comando en **Windows**:

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

7.1.3 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
```

7.1.4 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:

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.

***Tabla 7.1.** El comando JAR*

7.1.5 ¿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.

7.1.6 PROBLEMAS CON LOS FICHEROS JAR

Muchos de los problemas existentes con los ficheros **JAR** es por la asociación que hace el sistema operativo con el tipo de archivo. En la mayoría de sistemas basta con hacer doble click sobre el fichero **JAR** y automáticamente se ejecutará.

Si vemos que al hacer doble click el **JAR** no se ejecuta, tenemos que mirar si en el sistema la extensión **JAR** está asociada al *Java Runtime Environment* (**JRE**); también mirar si el fichero **JAR** tiene el logo de **Java** (muchos compresores asocian la extensión **JAR** y muchas otras al compresor, entonces en vez de ejecutar el archivo al hacer doble click se ejecutará siempre el compresor).

7.2 OTROS EMPAQUETADORES: EMPAQUETAMIENTO, INSTALACIÓN Y DESPLIEGUE

Aparte de los paquetes **JAR** convencionales, existen otras herramientas como los *wrappers*, instaladores y **JWS** para ejecutar y distribuir aplicaciones.

7.2.1 WRAPPERS

Los *wrappers* o **envoltorios** lo que hacen es envolver la aplicación en un ejecutable para una plataforma específica (**Windows**, **Mac OS X**, etc.). Algunas de las características de los *wrappers* son las siguientes:

- No hace falta extraer el **JAR** del ejecutable.
- Se pueden crear iconos para la aplicación.
- Puedes crear un *splash screen* (ventana de inicio mientras que se carga la aplicación).

- En el caso de no encontrar la versión de **Java** apropiada abre la página de descarga de la misma.
- Permite configurar ciertos parámetros de la aplicación como acceder al registro, variables de entorno, establecer parámetros de memoria, etc.

Algunos *wrappers* conocidos son por ejemplo **Lauch4J**, **jStart32** y **JSmoth**.

7.2.1.1 Launch4J

Figura 7.1. *Logo de Launch4j*

Página web oficial de *Launch4J*:

http://launch4j.sourceforge.net/

Launch4j es una herramienta multiplataforma para envolver aplicaciones **Java** en formato **JAR** dentro de ejecutables **Windows** de una manera intuitiva. Como se puede ver en la imagen siguiente, el ejecutable se puede configurar para que busque una versión de **JRE** o utilizar alguna ya empaquetada. Se puede configurar la aplicación con un icono o ventana *splash* lo cual hará que parezca más profesional.

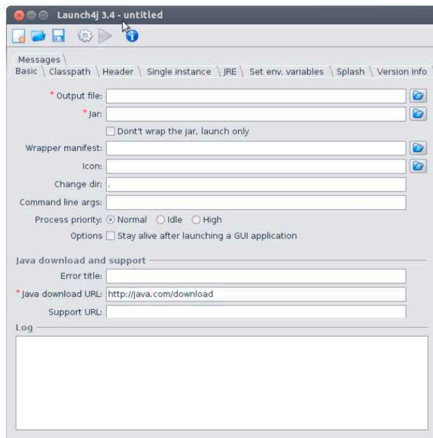

Figura 7.2. *Interfaz de Launch4j*

7.2.1.2 JSmooth

Figura 7.3. *Logo de JSmooth*

Página web oficial de *JSmooth*:

http://jsmooth.sourceforge.net/

JSmooth es un *wrapper* de aplicaciones **Java**. A diferencia con *Launch4j*, *JSmooth* no es multiplataforma, solamente funciona en **Windows** y al igual que *Launch4j*, puede buscar si tiene el equipo destino la **JVM** apropiada e instalará una, mostrará un mensaje de este hecho o bien te redirigirá a una página web para descargarla. JSmooth permite configurar las características del *wrapper*.

RECUERDA

Mac OS X tiene un **JAR bundler** el cual es un programa específico para **Mac** el cual crea **Bundles** o paquetes nativos de **OS X** con aplicaciones **Java**.

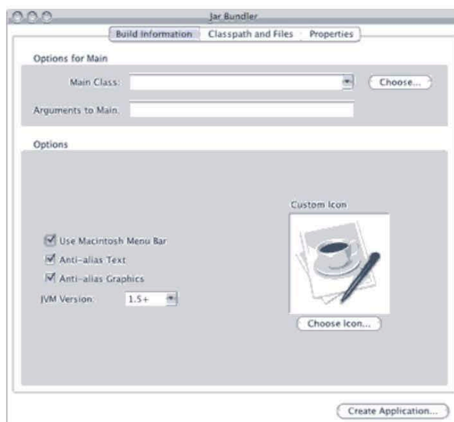

Figura 7.4. *Jar Bundler de Mac OS X*

7.2.2 LOS INSTALADORES

Los instaladores son programas que copian los ficheros de tu aplicación en el equipo del usuario y en ocasiones permiten crear un acceso directo para ejecutarlos.

Muchos instaladores al estar escritos en **Java** son multiplataforma (generalmente son ficheros **JAR**) y otros, al contrario, son dependientes de la plataforma.

7.2.2.1 IzPack

Figura 7.5. *Logo de IzPack*

Página web oficial de *IzPack*:

http://izpack.org/

IzPack es multiplataforma y se distribuye como un fichero **JAR**. Funciona en cualquier sistema operativo con una máquina **Java** SE™ 6+ o superior. Permite crear un instalador con accesos directos en el escritorio, modificación del registro de sistema o cambiar de usuario a administrador durante la instalación del sistema.

Permite configurar el instalador así como su apariencia y está publicado bajo licencia **Apache** *Software License*, Versión 2.0 lo cual implica que puedas modificarlo y adaptarlo a tus necesidades.

Figura 7.6. *Instalación de IzPack*

7.2.2.2 PackJacket

Figura 7.7. *Logo de PackJacket*

Página web oficial de PackJacket:

http://packjacket.sourceforge.net/

PackJacket es un interfaz visual para el proyecto *IzPack*. Desarrollado en **Java Swing**, ofrece un interfaz visual con el que se pueden crear instaladores multiplataforma y multilenguaje.

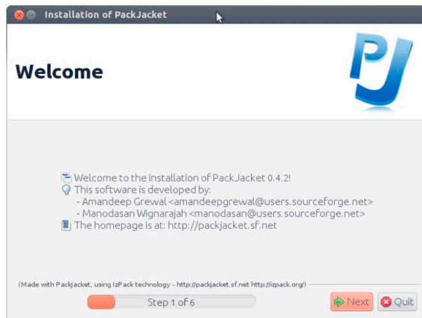

Figura 7.8. *Instalación de PackJacket*

7.2.3 JWS

JWS o *Java Web Start* es un *framework* desarrollado por **Sun Microsystems** (ahora **Oracle**) que permite ejecutar aplicaciones **Java** directamente de Internet. Una de las ventajas principales de **JWS** son las actualizaciones del programa. Los usuarios siempre están ejecutando la última versión puesto que la aplicación la bajan directamente de Internet.

A diferencia de los Applet los cuales corren dentro del navegador, las aplicaciones **JWS** corren fuera de él (aunque corren en el mismo *sandbox* que los Applet).

Para ejecutar la aplicación, el desarrollador crea un fichero **JNLP** que internamente es un fichero **XML** en el cual se especifican los recursos, los ficheros **JAR**, argumentos al programa, etc.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://myfpschool.com/Java/">
        <information>
                <title>MyFPSchool</title>
                <vendor>MyFPSchool</vendor>
                <homepage href="http://www.it-finance.com" />
                <description>MyFPSchool</description>
        </information>
        <resources>
                <j2se version="1.6.0+" href="http://java.sun.com/products/autodl/j2se" max-heap-size="800M" java-vm-args="-
noverify"/>
                <jar href="java/myfpschool.jar" main="true" download="eager" />
        </resources>
        <application-desc main-class="General.StartApplet">
        </application-desc>
</jnlp>
```

Figura 7.9. *Ejemplo de un fichero JNLP*

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes afirmaciones es falsa?:

a) El comando **jar tfv hola.jar holamundoswing.class** es para mostrar el contenido de un fichero **JAR**.

b) El formato **JAR** incluye muchas funcionalidades, entre otras el DNI electrónico.

c) Los ficheros **JAR** son multiplataforma.

d) El comando **jar cfv hola.jar holamundoswing.class** es para crear un fichero **JAR**.

2 ¿Cuál de las siguientes afirmaciones es falsa?:

a) Utilizaremos el comando java -cp hola.jar holamundoswing para ejecutar una aplicación en un **JAR**.

b) **Mac OS X** tiene un **JAR** bundler el cual es un programa específico para **Mac** y **Linux**, el cual crea Bundles o paquetes aplicaciones **Java**.

c) El manifiesto contiene información sobre los ficheros contenidos en el fichero **JAR**.

d) Utilizaremos el comando **jre -cp fichero_jar.jar Clase_main** para ejecutar una aplicación en un **JAR**.

8 DOCUMENTACIÓN DE APLICACIONES

La documentación es todo aquel conjunto de manuales impresos o en formato digital y cualquier otra información descriptiva que explique una aplicación informática o programa.

8.1 HERRAMIENTAS DE DOCUMENTACIÓN: CARACTERÍSTICAS

Las aplicaciones o programas tienen que estar perfectamente documentados, de lo contrario sería muy difícil mantener el código. En **Java**, la documentación del código se escribe dentro del mismo, lo cual es verdaderamente útil. **Java**, además tiene una herramienta que se llama **Javadoc** la cual extrae los textos y comentarios del **código fuente** y los transforma en páginas web (formato **HTML**).

Para escribir comentarios que los interprete **Javadoc** hay que insertarlos entre los caracteres /** y */. Dentro de estos caracteres especiales ya podemos escribir en **HTML** utilizando sus etiquetas y utilizando también otras etiquetas específicas como:

- **@author**: identifica al autor del código.
- **@version**: identifica la versión del código.
- **@since**: especifica la fecha de creación del código.
- **@param <nombre>**: describe el parámetro "nombre". Una línea por cada parámetro.
- **@return**: especifica el valor de retorno del método. Si la función devuelve *void* no hay que especificarlo.
- **@deprecated**: si la clase, método o *ítem* documentado está obsoleto.
- **@see**: utilizado si se quiere dirigir al lector a otro apartado.
- **@link**: utilizado si se quiere dirigir al lector al recurso especificado por la **URL**.

Veamos un ejemplo de documentación para el programa de colas visto en el segundo capítulo. Tenemos el código que se detalla a continuación:

```
/**
 * clase cola
 * @author Juan Carlos
 * @version 1.0
 * @since 21/04/2014
 * <br>
 * <p>Esta clase corresponde a un ejemplo de implementaci&oacute;n de una estructura
 * din&aacute;mica como es una cola.</p>
 */
public class Cola {
```

¿SABÍAS QUE?

Para generar la documentación de la clase anterior he ejecutado desde línea de comandos lo siguiente:

```
javadoc Cola.java
```

Tras ejecutar **Javadoc**, el cual se puede ejecutar desde el propio **IDE** (**NetBeans** o **Eclipse**) o desde línea de comandos (si utilizas **Geany** deberás ejecutarlo de esta manera), obtenemos una serie de páginas web con la documentación de la aplicación.

Package Class Tree Deprecated Index Help
Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Class Cola

java.lang.Object
Cola

```
public class Cola
extends java.lang.Object
```

clase cola

Since:

21/04/2014

Esta clase corresponde a un ejemplo de implementación de una estructura dinámica como es una cola.

Constructor Summary

Constructors

Figura 8.1. *Detalle de la generación HTML de Javadoc*

Herramientas para generación de ayudas: **JavaHelp**

Para que una aplicación tenga un menú de ayuda integrado en la propia aplicación generalmente se utiliza JavaHelp. JavaHelp es una extensión de Java que va a permitir implementar ventanas de ayuda dentro de Java. Estas ventanas de ayuda se basan en ficheros XML y HTML que son los que se mostrarán al usuario.

8.2 DOCUMENTACIÓN DE UNA APLICACIÓN

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).

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. La fase de planificación y estimación son las más complejas de un proyecto. Para esto se necesita gente con experiencia tanto en la elaboración de proyectos como en las plataformas en las que se va a realizar el mismo. De esta fase surgen muchos documentos tanto de planificación (general y detallada) como documentos de estimaciones en el que se incluyen datos económicos, posibles soluciones al problema y sus costes, etc. Son documentos que se realizan a alto nivel y se acuerdan con la dirección de la empresa o las personas responsables del proyecto. En estas fases iniciales se suelen tomar decisiones que a veces afectan a todas las demás fases del proyecto, con lo cual tiene que estar todo bien documentado, detallado y soportado por datos concretos.

- **Análisis**: en esta fase se analiza el problema. Consiste en recopilar, examinar y formular los requisitos del cliente y analizar cualquier restricción que se pueda aplicar. Todas las entrevistas con el cliente por lo tanto tienen que estar registradas en documentos y generalmente esos documentos se consensuan con el cliente y desde mi punto de vista algunos tienen hasta carácter contractual. Yo, como jefe de desarrollo, en algunos proyectos he hecho firmar al cliente un documento de requisitos de la aplicación en el cual mi equipo se compromete a realizar las especificaciones indicadas por el cliente y también el cliente se compromete a no variar sus necesidades hasta por lo menos terminar una primera *release*. Como puedes ver, es un documento que obliga a ambas partes a cumplir con lo acordado y en muchas ocasiones establece un marco de relación entre cliente y desarrollador.

- **Diseño**: esta fase consiste en determinar los requisitos generales de la arquitectura de la aplicación y dar una definición precisa de cada subconjunto de la aplicación. En esta fase los documentos ya son más técnicos. Se suelen crear dos documentos de diseño, uno más genérico en el que se tiene una visión de la aplicación más general y otro detallado en el que se profundizará en los detalles técnicos de cada módulo concreto del sistema. Estos documentos los realizarán los analistas junto con la supervisión del jefe de proyecto.

- **Codificación o implementación**: esta fase consiste en la implementación del *software* en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño. Durante esta fase se crea documentación muy detallada en el que se incluye código. Aunque mucho código se suele comentar en el mismo programa, también se tienen que generar documentos donde se indica por ejemplo para cada función las entradas, salidas, parámetros, propósito, módulos o librerías donde se encuentra, quién la ha realizado, cuándo, las revisiones que se han realizado de la misma, etc. Como puedes ver, el detalle es máximo teniendo en cuenta que ese código en un futuro va a tener que ser mantenido por la misma o seguramente otra persona y toda información que pueda recibir a veces es poca.
- **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. Como se ha visto anteriormente, las pruebas son de todo tipo. Yo clasificaría la documentación de las pruebas en dos bloques diferentes. Existen unas **pruebas funcionales** en las que se prueba que la aplicación hace lo que tiene que hacer con las funciones acordes a los documentos de especificaciones que se establecieron con el cliente, y esas pruebas se deberían realizar con el cliente delante. Mientras que se están realizando las pruebas se tienen que hacer todo tipo de anotaciones para luego plasmarlas en un documento en el que tendrá que darle el visto bueno el cliente (que por ese motivo estuvo en las pruebas). En ese documento se van detallando fallos tanto de la propia aplicación como modificaciones a la misma si no cumple con las especificaciones iniciales. En el caso que haya discrepancia se suele consultar documentación anterior para que no se incluyan en este punto funcionalidades nuevas o variaciones de las funcionalidades. En otro documento aparte se detallarán los resultados de las **pruebas técnicas** realizadas. A estas pruebas ya no hace falta que asista el usuario o cliente puesto que son de carácter meramente técnico. En estas pruebas se harán cargas reales, se someterá la aplicación y el sistema a estrés, etc.
- **Explotación**: en esta fase se instala el *software* en el entorno real de uso y se trabaja con él de forma cotidiana. Generalmente es la fase más larga y suelen surgir multitud de incidencias, nuevas necesidades, etc. Toda esta información se suele detallar en un documento en el que se documentan los errores o fallos detectados intentando ser lo más explícito posible puesto que luego los programadores y analistas deben de revisar estos fallos o *bugs* y darle la mejor solución posible. También surgirán otras necesidades que se van a ir detallando en un documento y que pasarán a realizarse en operaciones de mantenimiento.
- **Mantenimiento**: en esta fase se realizan todo tipo de procedimientos correctivos (corrección de fallos) y actualizaciones secundarias del *software* (mantenimiento continuo) que consistirán en adaptar y evolucionar las aplicaciones. Para realizar las labores de mantenimiento hay que tener siempre delante la documentación técnica de la aplicación. Sin una buena documentación de la aplicación, las labores de mantenimiento son muy difíciles y su garantía en ese caso sería poca. Todas las operaciones de mantenimiento tienen que estar documentadas porque se tiene que saber quién ha realizado la operación, qué ha hecho y cómo. También tienen que estar documentadas porque deberían probarse por otra persona distinta al programador.

En cada una de estas fases se generan uno o más documentos. En ningún proyecto es viable comenzar la codificación sin haber realizado las fases anteriores porque eso equivaldría a un desastre absoluto. Además, la documentación debe 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.

TEST DE CONOCIMIENTOS

1. ¿Cuál de las siguientes afirmaciones es falsa?:
   - a) En **Java** la documentación del código se escribe dentro del mismo.
   - b) En el manual de instalación se explica paso a paso los requisitos y cómo se instala y pone en funcionamiento la aplicación.
   - c) En la fase inicial de un proyecto, los documentos que se realizan son a alto nivel y se acuerdan con la dirección de la empresa.
   - d) En **Javadoc**, utilizaremos **@deprecated**. Si la clase, método o item no está documentado.

2. ¿Cuál de las siguientes afirmaciones es falsa?:
   - a) La documentación es un proceso que comienza desde el principio del proyecto y es algo que nunca termina.
   - b) En la fase de explotación se instala el *software* en el entorno real de uso.
   - c) Para escribir comentarios que los interprete **Javadoc** hay que insertarlos entre los caracteres /** y */.
   - d) **Javado**c siempre se ejecuta desde línea de comandos.

3. ¿Cuál de las siguientes afirmaciones es falsa?:
   - a) La fase de diseño consiste en recopilar, examinar y formular los requisitos del cliente y diseñar cualquier restricción que se pueda aplicar.
   - b) En la fase de diseño los documentos son más técnicos que en la fase de análisis.
   - c) Para que una aplicación tenga un menú de ayuda integrado en la propia aplicación generalmente se utiliza **JavaHelp**.
   - d) Para realizar las labores de mantenimiento hay que tener siempre delante la documentación técnica de la aplicación.

4. ¿Cuál de las siguientes afirmaciones es falsa?:
   - a) Es necesario que el usuario asista a todas las pruebas de la aplicación.
   - b) La fase de codificación consiste en la implementación del *software* en un lenguaje de programación.
   - c) En la descripción funcional de una aplicación se describe el funcionamiento del sistema.
   - d) En la fase de mantenimiento se realizan todo tipo de procedimientos correctivos.

Solucionario de los test de conocimientos

- CAPÍTULO 1: METODOLOGÍA DE LA PROGRAMACIÓN

  1-A 2-B 3-D 4-A 5-B 6-C

- CAPÍTULO 2: ESTRUCTURA DE DATOS

  1-C 2-C 3-C

- CAPÍTULO 3: PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS

  1-D 2-D

- CAPÍTULO 4: INTERFACES Y ENTORNOS GRÁFICOS

  1-C 2-C 3-D

- CAPÍTULO 5: ACCESO A BASES DE DATOS Y OTRAS ESTRUCTURAS

  1-D 2-C 3-B 4-A

- CAPÍTULO 6: PRUEBAS

  1-C 2-C 3-A 4-B

- CAPÍTULO 7: HERRAMIENTAS DE GENERACIÓN DE PAQUETES

  1-B 2-B

- CAPÍTULO 8: DOCUMENTACIÓN DE APLICACIONES

  1-D 2-D 3-A 4-A

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

Índice Alfabético

A

Accesibilidad, 94
Algoritmo, 35
Ámbito de una variable, 21
API, 100
Applets, 15
Arquitectura en dos niveles, 101
Arquitectura en tres niveles, 102
Arrays, 41
AWT, 79

B

Bases de datos, 100
Benchmark, 127
Beta testing, 122
Break, 33
Breakpoint, 67
Bucles, 30
Bytecode, 15

C

Cadenas de caracteres, 44
Caja blanca, 122
Caja negra, 122
Casos de pruebas, 124
Centralizados, 69
Ciclo del *software*, 123
Ciclo de vida clásico, 144
Código máquina, 14
Colas, 56
Comentarios, 16
Compiladores e intérpretes, 15
Concurrencia, 111
Constantes, 19
Control de excepciones, 33
Control de versiones, 69
Cursores, 113

D

Depuración, 67
Dictador y tenientes, 72
Distribuidos, 70
Documentación, 142

E

Eclipse, 66
Entorno Integrado de Desarrollo, 64
Errores de programación, 120
Estructuras condicionales, 28
Estructuras de datos, 40
Estructuras dinámicas, 50
Estructura secuencial, 27
Estructuras estáticas, 40
Estructuras iterativas, 30
Expresión, 27

F

Factorial, 36
FIFO, 56

G

Geany, 64
Gestor de integración, 71
git, 69

I

IDE, 64
Instaladores, 132
Interfaces, 78
IzPack, 137

J

JAR, 132
JAR bundler, 136
Java Development Kit, 64

Javadoc, 142
JavaHelp, 143
Java Web Start, 138
JDBC, 102, 113
JDK, 64
JFC, 79
JSmooth, 136
JWS, 132, 138

L

Launch4J, 135
Layout Manager, 82
lenguaje máquina, 15
Listener, 86
Literales, 20
Look and feel, 84

M

Main, 17
Manual de instalación, 146
Manual de usuario, 146
Manual técnico, 146
Matrices, 44
Muhammad ibn Musa Al-Khowarizmi, 35
Multiplataforma, 132

N

NetBeans, 66
Nodo, 50

O

Objeto statement, 108
Operadores aritméticos, 22
Operadores de bits, 25
Operadores lógicos, 24
Operadores relacionales, 23
Operadores unitarios, 25

P

PackJacket, 138
Palabras clave, 20
Persistencia, 112
Pilas, 53
Pop, 53
Precedencia de operadores, 26
Private, 61
Programa, 14
Programación estructurada, 15
Prueba de estabilidad, 127
Prueba de estrés, 122
Pruebas de carga, 127
Pruebas de estrés, 127
Pruebas de integración, 122
Pruebas de interfaces, 122
Pruebas de rendimiento, 127
Pruebas de software, 120
Pseudocódigo, 35
Public, 61
Puntero, 40
Push, 53

R

Recursividad, 36
Repositorio, 72
ResulSet, 111
Return, 33
Reutilización, 67

S

Scope, 21
SCV, 70
Servlets, 15
SGBD, 100
Sistemas de control de versiones, 69
SQL, 108
SQLException, 104
String, 45
Subversion, 74
Swing, 78, 79

T

Tester, 121
Testing, 120
Tipos abstractos de datos, 61
Tipos de datos, 17

U

Ubuntu Linux, 64
Usabilidad, 93

V

Variable, 20
Vectores, 41
Versión alpha, 122
Versión beta, 122
Visibilidad, 21

W

Workflow, 70
Workflow centralizado, 71
Wappers, 132, 134

X

XAMPP, 102

CERTIFICADO DE PROFESIONALIDAD

Programación en Lenguajes Estructurados

La presente obra está dirigida a los estudiantes del certificado de profesionalidad **Programación en Lenguajes Estructurados de Aplicaciones de Gestión**, en concreto al módulo formativo **Programación en lenguajes estructurados** y a toda aquella persona que quiera aprender a programar de forma estructurada con Java.

Los contenidos incluidos en este libro abarcan conceptos muy interesantes como la programación estructurada, las estructuras de datos estáticas y dinámicas, el diseño e implementación de interfaces de usuario, las pruebas del software, la documentación de los programas y sistemas, el acceso a bases de datos, etc.

Los capítulos incluyen notas, esquemas y ejemplos, con el propósito de facilitar la asimilación de los conocimientos tratados. Cuando termine de estudiar esta obra estará capacitado para empezar a desarrollar programas en Java, que es uno de los lenguajes con más futuro en la actualidad.

WWW Desde **www.ra-ma.es** podrá descargarse material adicional.