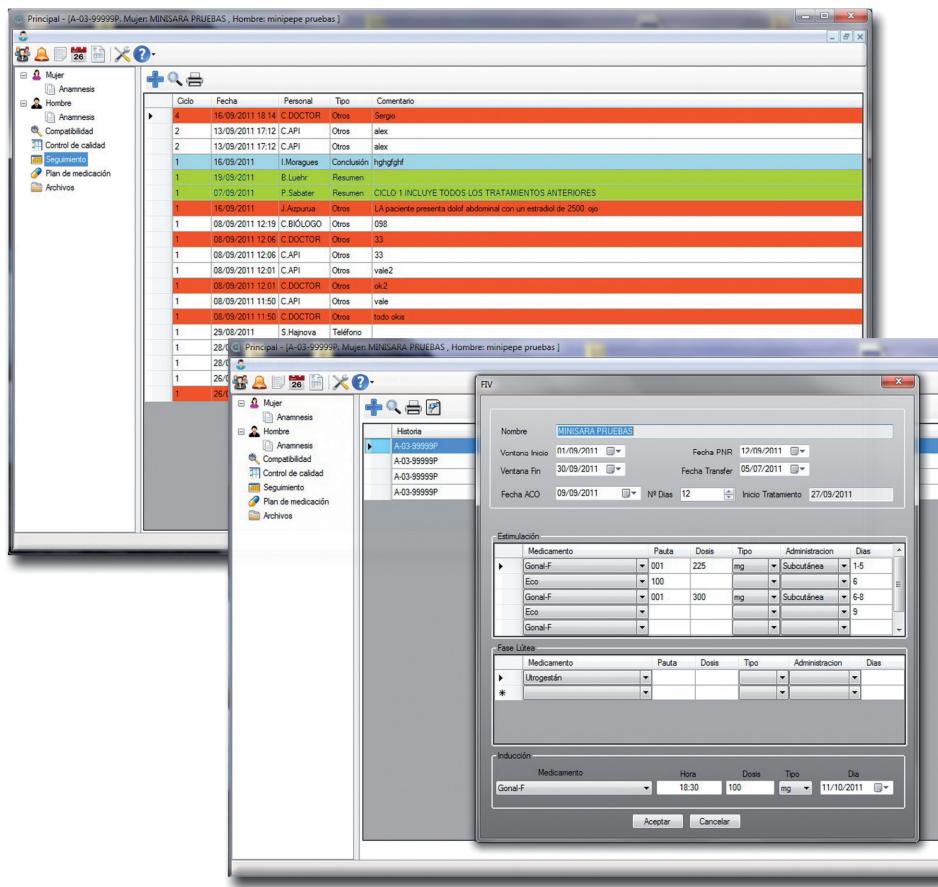


# CURSO DE DESARROLLO DE APLICACIONES WEB



Programación

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del *Copyright*, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, [www.cedro.org](http://www.cedro.org)) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN  
Centro de Estudios CEAC

COLABORADORES

*Realización:*  
ITACA (Interactive Training Advanced Computer Applications, S.L.)

*Elaboración de contenidos:*

Miguel Ángel Lozano  
Licenciado en Matemáticas  
Profesor de Informática y Coordinador de Calidad en el Instituto Ribera Baixa del Prat de Llobregat

*Actualización de contenidos:*

E-MAFE E-LEARNING SOLUTIONS, SL  
Carles Peiró  
Director IT & Shared Applications en General Cable

*Supervisión técnica y pedagógica:*

Departamento de Enseñanza de Centro de Estudios CEAC

*Coordinación editorial:*

Departamento de Producto de Centro de Estudios CEAC

© Planeta DeAgostini Formación, S.L.U.  
Barcelona (España), 2016

Segunda edición: marzo de 2018

ISBN: 978-84-9063-804-0 (Obra completa)  
ISBN: 978-84-9128-793-3 (Programación)

Depósito Legal: B 820-2018

Impreso por:  
QPPRINT  
C/ Comadrán, 7 Nave C4  
08210 Barberà del Vallès  
Barcelona

Printed in Spain  
Impreso en España

# INTRODUCCIÓN AL MÓDULO

---

El módulo *Programación* del ciclo formativo **Desarrollo de aplicaciones web** es la base necesaria, aunque no suficiente, para el desarrollo de aplicaciones.

El camino a recorrer en este módulo pasa por una serie de unidades que nos irán introduciendo poco a poco, pero de manera intensa, en el mundo de la programación. En concreto, se utilizará el lenguaje de programación Java, uno de los más empleados para el desarrollo de aplicaciones. También se harán comparaciones con C++, lenguaje similar tanto en la sintaxis como en la estructura.

En programación se utilizan tres metodologías, cada una de las cuales incluye y mejora la anterior: programación estructurada, modular y orientada a objetos.

En el aprendizaje de cualquier lenguaje de programación se siguen los pasos que iremos viendo a lo largo de las unidades de este módulo. Comenzamos con una primera unidad general, explicando los **conceptos** más significativos de los programas.

Se continúa con una introducción al mundo de los objetos, unas estructuras utilizadas en la programación para acercarnos al mundo real.

En la siguiente unidad, se describe la **sintaxis** del lenguaje, cómo se codifica en Java las instrucciones que son comunes a todos los lenguajes de programación, con especial énfasis en las estructuras de control.

Después pasamos a ver las **clases**, es decir, la definición del tipo de datos de los objetos, tanto una introducción como la utilización avanzada. Y por supuesto, en algún momento habrá datos que se guarden, en ficheros, en bases de datos relacionales y en bases de datos orientadas a objetos.

Al final, habremos aprendido a desarrollar **aplicaciones en Java**, tanto con diálogo por consola como con interfaz gráfica, tanto con el manejo de ficheros como de bases de datos.

## Esquema de contenido

### 1. Identificación de los elementos de un programa informático

- 1.1 Estructura y bloques fundamentales
- 1.2 Utilización de los entornos integrados de desarrollo
- 1.3 Proyectos y soluciones
- 1.4 Datos: naturaleza y tipos
- 1.5 Operadores
- 1.6 Expresiones
- 1.7 Conversiones de tipo
- 1.8 Comentarios

### 2. Utilización de objetos

- 2.1 Características de los objetos
- 2.2 Instanciación de objetos
- 2.3 Utilización de métodos y propiedades
- 2.4 Programación de la consola: entrada y salida de información
- 2.5 Utilización de métodos estáticos
- 2.6 Parámetros y valores devueltos
- 2.7 Librerías de objetos
- 2.8 Constructores
- 2.9 Destrucción de objetos y liberación de memoria

### 3. Uso de estructuras de control

- 3.1 Estructuras de selección
- 3.2 Estructuras de repetición
- 3.3 Estructuras de salto
- 3.4 Control de excepciones
- 3.5 Prueba y depuración
- 3.6 Documentación

### 4. Desarrollo de clases

- 4.1 Estructura y miembros de una clase
- 4.2 Creación de atributos
- 4.3 Creación de métodos
- 4.4 Creación de constructores
- 4.5 Encapsulación, ocultamiento y visibilidad
- 4.6 Utilización de clases y objetos
- 4.7 Utilización de clases heredadas
- 4.8 Empaquetado de clases

## **5. Lectura y escritura de la información**

- 5.1 Concepto de flujo
- 5.2 Tipos de flujos. Flujos de bytes y de caracteres
- 5.3 Flujos predefinidos
- 5.4 Clases relativas a flujos
- 5.5 Utilización de flujos
- 5.6 Entrada desde teclado
- 5.7 Salida a pantalla
- 5.8 Aplicaciones del almacenamiento de información en ficheros
- 5.9 Ficheros de datos. Registros
- 5.10 Apertura y cierre de ficheros. Modos de acceso
- 5.11 Escritura y lectura de información en ficheros
- 5.12 Almacenamiento de objetos en ficheros. Persistencia. Serialización
- 5.13 Utilización de los sistemas de ficheros
- 5.14 Creación y eliminación de ficheros y directorios
- 5.15 Creación de interfaces gráficos de usuario utilizando asistentes y herramientas del entorno integrado
- 5.16 Interfaces
- 5.17 Concepto de evento
- 5.18 Creación de controladores de eventos
- 5.19 Generación de programas en entorno gráfico

## **6. Aplicación de las estructuras de almacenamiento**

- 6.1 Estructuras
- 6.2 Creación de arrays
- 6.3 Inicialización
- 6.4 Arrays multidimensionales
- 6.5 Cadenas de caracteres
- 6.6 Listas
- 6.7 Colecciones

## **7. Utilización avanzada de clases**

- 7.1 Encapsulación de datos
- 7.2 Herencia
- 7.3 Superclases y subclases
- 7.4 Clases y métodos abstractos y finales
- 7.5 Sobreescritura de métodos
- 7.6 Constructores y herencia
- 7.7 Acceso a métodos de la superclase
- 7.8 Polimorfismo

## **8. Mantenimiento de la persistencia de los objetos**

- 8.1 Bases de datos orientadas a objetos
- 8.2 Características de las bases de datos orientadas a objetos
- 8.3 Instalación del gestor de bases de datos
- 8.4 Creación de bases de datos
- 8.5 Tipos de datos básicos y estructurados
- 8.6 El lenguaje de definición de objetos
- 8.7 Mecanismos de consulta
- 8.8 El lenguaje de consultas: sintaxis, expresiones, operadores
- 8.9 Recuperación, modificación y borrado de información
- 8.10 Tipos de datos objeto; atributos y métodos
- 8.11 Herencia
- 8.12 Constructores
- 8.13 Tipos de datos colección

## **9. Gestión de bases de datos relacionales**

- 9.1 Establecimiento de conexiones
- 9.2 Recuperación de información
- 9.3 Utilización de asistentes
- 9.4 Manipulación de la información
- 9.5 Mecanismos de actualización de la base de datos
- 9.6 Ejecución de consultas sobre la base de datos

# 1. IDENTIFICACIÓN DE LOS ELEMENTOS DE UN PROGRAMA INFORMÁTICO

El concepto *programación* consiste en definir una serie de directrices con las que se obtiene un resultado que responde a la resolución de un problema. Un programa informático sigue exactamente esta premisa. A través de los lenguajes de programación, podremos definir las directrices que indicarán al ordenador, de un modo comprensible para su procesador, cómo llegar a la resolución de un problema expuesto.

Existen numerosos lenguajes de programación, todos con sus propias características y, por lo tanto, destinados a resolver diferentes tipos de problemas. Cada lenguaje está especializado en un ámbito determinado de la informática, sin embargo todos ellos parten de una base común.

Según su complejidad, podemos distinguir entre lenguajes de alto nivel (C++ o Java) y de bajo nivel (*assembler*). Los de alto nivel, permiten desarrollar programas simplificando el modo en que dirigimos las instrucciones al ordenador, mientras que los de bajo nivel se basan en instrucciones dirigidas directamente al hardware interno del ordenador, por lo que presentan una complejidad de control y organización muy elevada.

Tanto por su mayor facilidad de aprendizaje y manejo como por su uso más extendido, este módulo está centrado en los lenguajes de alto nivel.

A lo largo de los siguientes apartados, aprenderemos a aplicar las instrucciones comunes a todos los lenguajes de programación, lo que nos permitirá diseñar un programa informático.

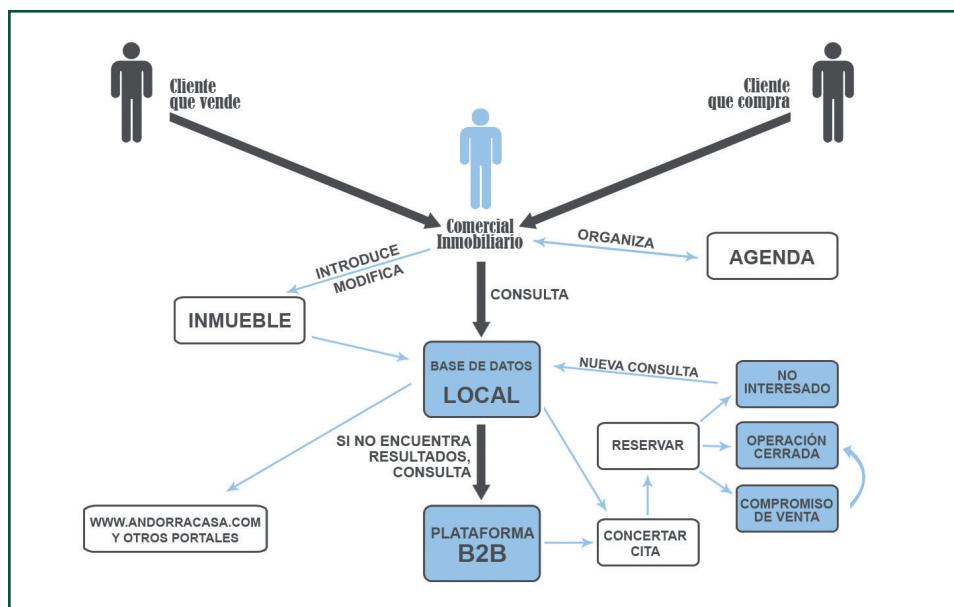
## 1.1 Estructura y bloques fundamentales

Un programa informático es una secuencia lógica de instrucciones escritas en un lenguaje de programación determinado que, al ser interpretadas por un ordenador, dan solución a un problema mediante la manipulación de un conjunto de datos. Una vez definidas, dichas instrucciones dan lugar a lo que denominamos **código fuente**.

Todo lenguaje de programación está sujeto a una serie de normas sintácticas y estructurales propias. La elección del lenguaje que vayamos a usar dependerá de los requerimientos del problema a resolver. Cualquier programa informático, in-

dependientemente del lenguaje en el que esté escrito, puede dividirse en dos bloques principales (Figura 1.1) que definen su estructura base:

- **Bloque de declaraciones.** Conjunto de datos proporcionados al programa informático destinados a ser manipulados a través de las instrucciones definidas en él. Dichos datos pueden ser de diferente origen y naturaleza.
- **Bloque de instrucciones.** Conjunto de acciones que usará el programa para manipular los datos proporcionados y obtener los resultados necesarios para la resolución del problema propuesto. El bloque de instrucciones, por tanto, debe resolver los siguientes problemas:
  - **Entrada.** Recogida de los datos externos al programa necesarios para resolver el problema.
  - **Transformación.** Conversión de los datos para conseguir el resultado.
  - **Salida.** Enviar a su destinatario la solución al problema, de modo que lo pueda entender.



**Figura 1.1**

Esquema de la estructura básica de un programa informático.

## 1.2 Utilización de los entornos integrados de desarrollo

El **entorno de desarrollo integrado** (IDE, *Integrated Development Environment*) es un programa informático que, de una manera atractiva para el usuario, ofrece un conjunto de herramientas y recursos necesarios para el desarrollo de aplicaciones informáticas en uno o más lenguajes de programación (Figura 1.2).

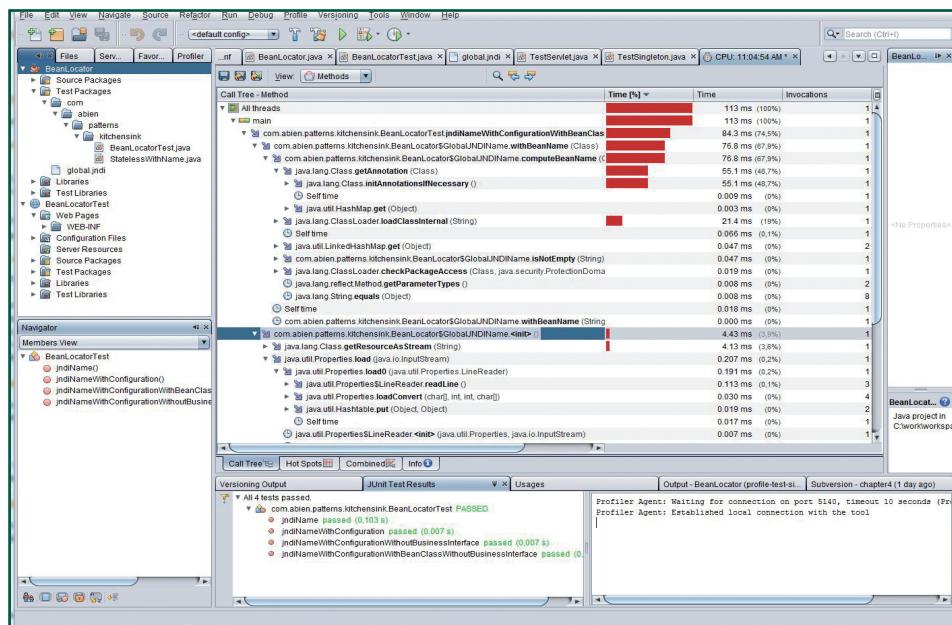
El objetivo de un IDE es facilitar el desarrollo de aplicaciones y programas informáticos empaquetando los recursos en un marco de trabajo estable y visual.

Cualquier IDE está formado por los siguientes bloques:

- **Editor de código.** Entorno sobre el que se escribe el código fuente de nuestro programa. Ofrece ayudas visuales, como el formato textual para resaltar los aspectos relevantes del código.
- **Compilador o intérprete.** Herramienta destinada a interpretar el código escrito y evaluar la sintaxis del código generado.
- **Depurador.** Detecta y solventa los errores que pueden surgir en la escritura del código. Permite, por ejemplo, ejecutar línea a línea, ver el valor de algunos datos, etc.
- **Constructor de interfaz gráfica.** Actúa como un conjunto de opciones visuales que permiten la comunicación entre el usuario y el ordenador a la hora de escribir e interpretar el código escrito.

## Recuerda

**Independientemente del lenguaje de programación que usemos, la estructura organizativa del código fuente será siempre la misma.**



**Figura 1.2**  
Interfaz de desarrollo de un IDE.

Algunos de los IDE más utilizados en Java son Eclipse y NetBeans, siendo este último el que se ha utilizado para desarrollar los ejemplos de este curso. Tanto Eclipse como NetBeans pueden ser usados en los lenguajes C y C++ si se instala un *plugin* adicional..

## 1.3 Proyectos y soluciones

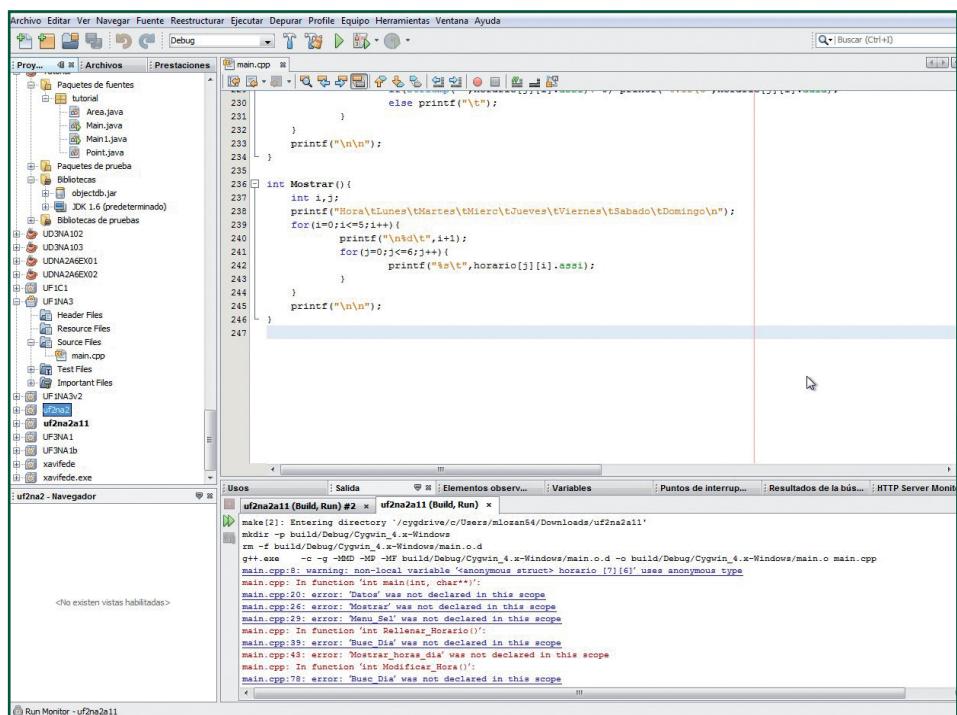
Durante el proceso de desarrollo de una aplicación entran en juego muchos factores y elementos, además del código fuente, todos importantes para el correcto funcionamiento.

## Para saber más

**Los lenguajes de programación más utilizados para el desarrollo de aplicaciones informáticas son C++ y Java. Se trata de lenguajes estables, versátiles y potentes, por lo que son adecuados para realizar gran variedad de programas.**

to del programa. Por esa razón deberán organizarse de un modo que facilite el acceso a ellos. Para tales efectos, los IDE ofrecen **soluciones**, como la creación de **proyectos**.

A efectos prácticos, un  **proyecto** es la organización de los archivos que intervienen en la ejecución de un programa (Figura 1.3). Cuando creamos un proyecto para nuestra aplicación, lo que en realidad estamos haciendo es definir un cuadro organizativo para todos los recursos que vayamos a necesitar. Así pues, podremos definir una carpeta para archivar las imágenes, otra para archivos de texto, otra para el resto de archivos con código necesario para la aplicación, etc. Algunas de las carpetas vendrán definidas por defecto, ya que irán destinadas a datos internos del programa sobre los que no deberemos actuar. Otras, en cambio, las podremos definir nosotros, bien durante la creación del proyecto o bien a medida que sea necesario.



**Figura 1.3**

Ejemplo de estructura de un proyecto para lenguaje de programación C++.

## 1.4 Datos: naturaleza y tipos

Todos los datos destinados a ser manipulados por un programa informático se almacenan en nuestro ordenador dentro de espacios de memoria determinados.

Según su naturaleza, podemos diferenciar tres clases de datos: variables, constantes y literales. Todos ellos deberán estar definidos en el bloque de declaraciones de nuestro programa.

### 1.4.1 Variables

Denominamos **variable** a todo aquel espacio de memoria reservado e identificado que, a lo largo de la ejecución del programa, puede ver alterado el valor que almacena (Figura 1.4).

Podemos usar una variable tanto para almacenar un valor destinado a ser manipulado, como para almacenar el resultado de una manipulación de datos. Para declarar una variable deberemos tener en cuenta los siguientes aspectos:

Toda variable:

- Deberá estar definida en el bloque de declaraciones de su ámbito (programa, clase, función, método, sentencia de control...).
- Deberá tener un nombre que la identifique de forma única en su ámbito.
- En la mayoría de los lenguajes se indicará el tipo de dato.
- Opcionalmente se le puede asignar un valor inicial.

Lenguaje	Declaración de variables
Java	String nombre = "Juan Pérez"; int velocidad = 120;
JavaScript	var nombre = "Juan Pérez"; var velocidad = 120;
C++	char nombre[] = "Juan Pérez"; int velocidad = 120;
PHP	\$var_nombre = "Juan Pérez"; \$var_velocidad = 120;

**Figura 1.4**

Ejemplo de declaración de variables definidas en diferentes lenguajes de programación.

### 1.4.2 Constantes

La **constante** es aquel espacio de memoria reservado e identificado mediante un nombre que almacenará un valor fijo e inalterable a lo largo de la ejecución del programa. Por lo general, una constante corresponde a un dato ubicado en la memoria principal del ordenador. Como ejemplo podríamos tomar el valor numérico **PI**, que podríamos asociar con el valor 3.141592.

Asimismo, y siguiendo las mismas reglas que se aplican a las variables, podemos crear nuestras propias constantes asignándoles los valores deseados. No obstante, y a diferencia de una variable, una constante nunca puede declararse sin asignarle un valor inicial.

## Recuerda

**Para mantener el código organizado, es imprescindible dotar a los datos con nombres que nos ayuden a reconocer cómo están destinados a ser manipulados.**

A continuación se indica cómo se declaran constantes en algunos lenguajes:

En C realmente es una directiva del compilador que hará que cambie la constante por su valor

```
#define PI 3.14159
```

En C++, aunque también se permite la forma anterior, la más correcta es:

```
const float PI=3.14159;
```

Y en Java

```
final float PI=3.14159;
```

### 1.4.3 Literales

## Recuerda

**Cada lenguaje de programación tiene una sintaxis propia para declarar los datos que intervendrán durante la ejecución de un programa. Por tanto, la manera de definirlos y de asignarles valores variará según el lenguaje que estemos usando.**

Una **literal** es un tipo de dato al que se le asigna un valor con un sentido estricto. Los compiladores informáticos tienen diferentes maneras de interpretar los datos que intervienen en la ejecución de un programa. Supongamos, por ejemplo, que queremos almacenar el valor numérico **100** en una variable. Al ejecutar el programa, según el lenguaje en el que estemos trabajando, este dato puede ser interpretado por el compilador como 4 en base binaria.

Para evitar este tipo de confusiones a la hora de ejecutar un programa, dichos lenguajes incluyen la posibilidad de definir los datos de un modo estricto, es decir, si en una literal almacenamos el numeral **100**, significará 100 exactamente, sin posibilidad de que sea interpretado de otro modo por el ordenador. Otro ejemplo de valor literal sería un dato destinado a almacenar un nombre, un conjunto de letras, o varias palabras. Toda letra o palabra hace referencia a un dato que no puede interpretarse de otra manera, se trata pues de un **dato literal** (Figura 1.5).

**Figura 1.5**  
Valores literales asignados a diferentes variables.

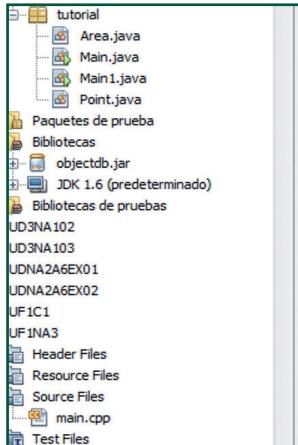
Tipo de literal	Definición en código
Literal en base decimal	long edad = 80L;
Literal de carácter	char letra_c ='c';
Literal de cadena de caracteres	string nombre = "Juan";

### 1.4.4 Tipos de datos

Según el tipo de datos que se defina para guardar un valor, el ordenador reservará un espacio de memoria y de tamaño para su correcto almacenaje. Veamos a continuación los tipos de datos principales:

- **Entero (int).** Valores numéricos positivos y negativos no decimales.
- **Real (float).** Valores numéricos positivos y negativos con parte decimal.
- **Carácter (char).** Valores alfanuméricos simples, es decir, una letra, un número o un símbolo, siempre que estos tengan una longitud unitaria.
- **Valores lógicos (boolean).** Se utilizan para datos que disponen sólo de dos posibles valores: verdadero (*true*) o falso (*false*).
- **Cadena de caracteres (String).** Valores alfanuméricos que pueden estar compuestos por más de un carácter. Un valor de tipo *string*, así como uno de tipo *char*, hace la función de una palabra, es decir, en caso de ser un número, éste no podrá actuar en operaciones matemáticas. Éste no es realmente un tipo de dato base, pero lo tienen muchos lenguajes de programación; C, no, pero sí Java.

Es importante definir el tipo del valor que queremos almacenar (Figura 1.6), pues de ello dependerá la cantidad de memoria utilizada por la máquina al ejecutar el programa.



```

1 #include <iostream>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <iostream>
5
6 #define PI 3.14
7
8
9 using namespace std;
10
11 int main(int argc, char *argv[]) {
12
13     int a = 5;
14     char c = 'd';
15     float importe = 3.5;
16
17     char nombre[20] = "Maria";
18
19     cout<<"El numero Pi es "<<PI;
20
21     return 0;

```

**Figura 1.6**  
Ejemplo de declaración de variables que almacenan distintos tipos de datos en lenguaje de programación C++.

### 1.5 Operadores

Un **operador** es aquel elemento que permite manipular los datos declarados en nuestro programa. El operador no es más que uno o más símbolos asociados a una determinada operación, como, por ejemplo, el operador + es la suma y el operador - , la resta. Existen operadores unarios, si sólo afectan a un operando; binarios, a dos, y ternarios.

## Para saber más

**El operador ++ y -- se pueden utilizar en forma preincremento (a++) o postincremento (++a). En el primer caso , primero evalua y después incrementa, y en el segundo caso al revés.**

**Ejemplo:**

```
int a=1;
int b,c;
b=a++;
//a y b valdrán 1
c=++b;
// c valdrá 2 y b 2
```

**Figura 1.7**

Tabla de los diferentes operadores aritméticos.

Existen diferentes clases de operadores.

- **Operador de asignación simple:**

- Asignación simple (=). Almacena el valor a la derecha del símbolo en la variable de la izquierda.

- **Operadores aritméticos binarios (Figura 1.7):**

- Suma (+). Suma entre valores.
- Resta (-). Resta entre valores.
- Multiplicación (\*). Multiplicación entre valores.
- División (/). Cociente de la división entre valores.
- Módulo (resto) (%). Resto de la división entre valores. Sólo para tipos enteros.

- **Operadores aritméticos unarios (Figura 1.7):**

- Incremento (++) . Aumenta el valor en una unidad.
- Decremento (--). Disminuye el valor en una unidad.
- Cambio de signo (-). Invierte el signo de un valor de positivo a negativo o viceversa.

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Suma	+	4 + 2	6
Resta	-	4 - 2	2
Multiplicación	*	4 * 2	8
División	/	4 / 2	2
Resto de división	%	7 % 2	1
Incremento	++	++2	3
Decremento	--	--3	2
Cambio de signo	-	-(2*3)	-6

- **Operadores de comparación (Figura 1.8):**

- Igualdad (==). Compara si dos datos contienen el mismo valor.
- Desigualdad (!=). Compara si dos datos contienen valores distintos.
- Menor que (<). Compara si el valor de la izquierda es menor que el de la derecha.
- Mayor que (>). Compara si el valor de la izquierda es mayor que el de la derecha.
- Menor o igual que (<=). Compara si el valor de la izquierda es menor o igual que el de la derecha.
- Mayor o igual que (>=). Compara si el valor de la izquierda es mayor o igual que el de la derecha.

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Igualdad	<code>==</code>	<code>2 == 2</code>	Verdadero ( <i>true</i> )
Desigualdad	<code>!=</code>	<code>2 != 2</code>	Falso ( <i>false</i> )
Menor que	<code>&lt;</code>	<code>2 &lt; 2</code>	Falso ( <i>false</i> )
Mayor que	<code>&gt;</code>	<code>3 &gt; 2</code>	Verdadero ( <i>true</i> )
Menor o igual que	<code>&lt;=</code>	<code>2 &lt;= 2</code>	Verdadero ( <i>true</i> )
Mayor o igual que	<code>&gt;=</code>	<code>1 &gt;= 2</code>	Falso ( <i>false</i> )

**Figura 1.8**

Tabla de los diferentes operadores de comparación.

#### • Operadores lógicos:

- Y (`&&`). La operación es cierta sólo si los dos valores que tiene a ambos lados son ciertos, y falsa en cualquier otro caso.
- O (`||`). La operación es cierta si cualquiera de los dos valores que tiene a ambos lados es cierto, y falsa sólo en caso de que ambos valores sean falsos.
- NO (`!`). Si el valor que tiene a la derecha es falso el resultado de la operación será cierto, si el valor es cierto el resultado será falso.

#### • Operadores de asignación compuestos (Figura 1.9):

Estos operadores permiten realizar una operación aritmética y la asignación evitando escribir dos veces el resultado de la variable que se verá modificada. Aunque nos permiten velocidad en la escritura, se desaconseja su uso inicialmente, ya que son menos legibles.

Expresión	Operación resultante
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>

**Figura 1.9**

Tabla de los diferentes operadores de asignación.

Todos los operadores tienen un orden de prioridad; es decir, en caso de haber más de uno, cuál de ellos evaluará primero. En caso de que tengamos que variar el orden de evaluación, o no recordemos las prioridades y queramos asegurar el orden, se usan los paréntesis de forma similar a los cálculos matemáticos. Por ejemplo:

```
float media;
float a,b;
media = (a + b) / 2.0;
```

## Para saber más

Algunos lenguajes de programación, como por ejemplo Java o C++, pueden manipular los datos a niveles muy bajos, incluso de bits, por lo que incluyen operadores específicos para tales efectos.

## 1.6 Expresiones

En programación, una expresión es una combinación de datos que, al ser evaluada por el ordenador con la ayuda de operadores, ofrece un resultado en función del valor y el tipo de los datos. El resultado obtenido podrá utilizarse para almacenarlo en un nuevo dato o para realizar una acción determinada según su valor.

Existen tres tipos básicos de expresiones de programación:

- **Aritméticas.** Hacen referencia a todas las operaciones matemáticas como sumas, restas y divisiones.
- **Relaciones.** Se usan para comparar el valor de dos datos y definir si estos son iguales, diferentes o, si siendo diferentes, uno es mayor o menor que el otro. Este tipo de expresiones sólo tienen dos resultados posibles: verdadero o falso (Figura 1.10).
- **Lógicas.** Sirven para comprobar si se cumplen dos o más premisas expuestas. Igual que las expresiones relacionales, este tipo de expresiones sólo tienen dos resultados posibles: verdadero o falso (Figura 1.10).

Expresiones relacionales			
Valor 1	Operador	Valor 2	Resultado
A	>	B	Verdadero/Falso
A	<	B	Verdadero/Falso
A	$\geq$	B	Verdadero/Falso
A	$\leq$	B	Verdadero/Falso
A	$\neq$	B	Verdadero/Falso
Expresiones lógicas			
Valor 1	Operador	Valor 2	Resultado
A	$\&\&$ (Conjunción)	B	Verdadero/Falso
A	$\ $ (Conjunción)	B	Verdadero/Falso

**Figura 1.10**

Tabla referencial de los distintos tipos de expresiones de programación con sus respectivos resultados.

## 1.7 Conversiones de tipo

Cada tipo de datos es almacenado de una manera diferente en el ordenador y permite realizar diferentes operaciones, pero a veces es necesario realizar operaciones sobre un dato que no se corresponde con su tipo. Por ejemplo, un valor numérico que el usuario ha introducido en forma de cadena debe ser tratado como un número para participar en una operación aritmética.

Las conversiones de tipo (en inglés cast) nos permiten transformar el valor que contiene un dato en un valor de tipo distinto, y puede ser implícita si el compilador o intérprete es capaz de realizarla automáticamente (habitual cuando pasamos de un valor particular a otro más general) o explícita si el compilador la pide expresamente.

Supongamos por un momento que, tras una operación matemática, como por ejemplo una división, el resultado obtenido aparece con una parte decimal que queremos desechar. Dicho valor estará almacenado en una variable de tipo *float*. Usando una conversión de tipo podremos conseguir que el valor numérico almacenado como tipo *float* pase a ser de tipo *int*, desechando así directamente su parte decimal.

Cada lenguaje de programación tiene sus propias herramientas para realizar los cambios de tipo. La más utilizada es el *cast*, que utilizan el lenguaje C y sus derivados, además de Java y PHP. Por ejemplo, *UnNumeroEntero = (int)UnNumeroDecimal;* es una sentencia que convierte el valor de la variable *UnNumeroDecimal* en un número entero y almacena el resultado en la variable *UnNumeroEntero*.

## 1.8 Comentarios

Un **comentario** es un texto descriptivo que añade información adicional dentro del código fuente de un programa. Los comentarios no son interpretados por el compilador del lenguaje a la hora de ejecutar el programa. Suelen utilizarse algunos símbolos para indicar el inicio del comentario y se acaban bien con el fin de línea, o bien con algún símbolo. El uso de los comentarios es muy extenso, pueden emplearse tanto para introducir datos aclaratorios dentro del código como para evitar que el programa active una cierta instrucción. Cada lenguaje de programación tiene una sintaxis propia para definir comentarios dentro del código (Figura 1.11).

### Recuerda

**Los comentarios no tienen ningún efecto durante la ejecución de un programa, pero siempre nos ayudarán a mantener un código organizado y comprensible.**

Lenguaje	Comentarios
Java	//Comentario de línea /* Comentario de bloque */
JavaScript	//Comentario de línea /* Comentario de bloque */
C/C++	//Comentario de línea /* Comentario de bloque */
SQL	//Comentario de línea -- Otro comentario de línea /* Comentario de bloque */
PHP	//Comentario de línea # Otro comentario de línea /* Comentario de bloque */
Lua	--Comentario de línea --[ Comentario de bloque ]--

**Figura 1.11**

Ejemplos de comentarios de código según diferentes lenguajes de programación.

## Resumen

En esta unidad hemos definido las normas y comportamientos básicos que se aplican a todo lenguaje de programación. A la hora de diseñar un programa informático deberemos recordar los siguientes pasos básicos a seguir: identificación de un problema, definición de directrices, obtención de resultados, depuración de errores y resolución del problema.

El **entorno de desarrollo integrado** (IDE, *Integrated Development Environment*) nos facilitará las herramientas necesarias para comunicarnos con el ordenador a través de instrucciones escritas en un lenguaje de programación determinado.

Para que nuestro programa sea ágil y estable, deberemos tener en cuenta la cantidad de datos que necesitaremos y el modo en que los almacenaremos en la memoria del ordenador. Los diferentes tipos de datos nos ayudarán a crear parcelas de almacenaje para cada uno de los datos que usemos.

Los operadores, las expresiones y la conversión de datos son herramientas que nos permitirán manipular los datos a lo largo de nuestro programa con el fin de proporcionarnos una resolución al problema planteado.

Los lenguajes de programación ofrecen, además, ciertos recursos para hacer que nuestro código sea más versátil y comprensible. Recordemos que herramientas como los comentarios no influyen en el resultado final del programa, pero son imprescindibles para mantener organizado el código de éste.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Cualquier programa informático, independientemente del lenguaje en el que esté escrito, puede dividirse en dos bloques principales que definen su estructura base: el bloque de declaraciones y el bloque de instrucciones.
2. El entorno de desarrollo integrado (IDE) es un programa informático que ofrece un conjunto de herramientas y recursos necesarios para el desarrollo de aplicaciones informáticas en un solo lenguaje de programación.
3. Independientemente del lenguaje de programación que usemos, la estructura organizativa del código fuente será siempre la misma.
4. Según su naturaleza, podemos diferenciar tres clases de datos: variables, constantes y literales. Todos ellos deberán estar definidos en el bloque de declaraciones de nuestro programa.
5. Toda variable deberá estar definida en el bloque de instrucciones de su ámbito, deberá tener un nombre que la identifique de forma única en su ámbito, en la mayoría de los lenguajes se indicará el tipo de dato y, opcionalmente, se le puede asignar un valor inicial.
6. Los compiladores informáticos tienen una manera determinada de interpretar los datos que intervienen en la ejecución de un programa.
7. Un valor de tipo *string*, así como uno de tipo *char*, hace la función de una palabra, es decir, en caso de ser un número, este no podrá actuar en operaciones matemáticas.

**Completa las siguientes afirmaciones:**

8. Los lenguajes de alto nivel, permiten desarrollar \_\_\_\_\_ simplificando el modo en que dirigimos las instrucciones al ordenador, mientras que los de bajo nivel se basan en instrucciones dirigidas directamente al \_\_\_\_\_ interno del ordenador, por lo que presentan una complejidad de \_\_\_\_\_ y \_\_\_\_\_ muy elevada.

9. Cada lenguaje de programación tiene una \_\_\_\_\_ propia para declarar los datos que intervendrán durante la ejecución de un \_\_\_\_\_. Por tanto, la manera de definirlos y de asignarles valores variará según el \_\_\_\_\_ que estemos usando.
10. Algunos lenguajes de programación, como por ejemplo \_\_\_\_\_ o \_\_\_\_\_, pueden manipular los datos a niveles muy \_\_\_\_\_, incluso de bits, por lo que incluyen \_\_\_\_\_ específicos para tales efectos.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 2. UTILIZACIÓN DE OBJETOS

A lo largo de esta unidad vamos a estudiar las bases de la programación orientada a **objetos** (POO o, en sus siglas en inglés, OOP, *Object-oriented programming*).

La programación orientada a objetos debe tomarse como una filosofía, un modelo de programación. A continuación, estudiaremos los conceptos básicos que se aplican a la POO desde un punto de vista general, sin particularizar en ningún lenguaje de programación específico.

Es muy importante señalar que cuando hacemos referencia a la programación orientada a objetos no estamos hablando de características concretas añadidas a un lenguaje de programación, sino que hablamos de una nueva forma de pensar acerca del proceso de descomposición del problema y de desarrollo de soluciones.

La POO propone una serie de recursos basados en entidades del mundo real para la construcción de estructuras y soluciones. La POO surge como un intento que pretende ayudar a controlar la complejidad inherente al desarrollo de aplicaciones informáticas.

En POO, un objeto es la abstracción de un elemento real, al que se dota de atributos representados por sus características o propiedades, y que representan las acciones que realiza. Si aprendemos el uso de los objetos seremos capaces de crear códigos más simplificados, reutilizables y que propiciarán el mejor funcionamiento de nuestro programa.

El conocimiento de los conceptos y procedimientos que se desarrollan en esta unidad, es una buena base de partida para el aprendizaje de cualquier lenguaje de programación orientado a objetos. En cada uno de ellos, las instrucciones pueden ser diferentes, pero aunque cambie el lenguaje, los conceptos a manejar serán prácticamente los mismos.

### 2.1 Características de los objetos

En programación, un **objeto** es una entidad provista de un conjunto de **propiedades** (datos) y de **comportamientos** (métodos) que reaccionan según determinados eventos. Un objeto puede ser, tanto la representación informática de un concepto del mundo real, como la de objetos internos de un programa informático.

Los objetos tienen tres características principales:

- **Estado.** Hace referencia al conjunto de **datos** (propiedades, variables) que pertenecen al objeto, a los que podremos asignar valores.
- **Comportamiento.** Viene definido por el conjunto de **métodos** (funciones) que contiene un objeto. Hace referencia a todas las operaciones que podremos realizar con él.
- **Identidad o nombre.** Es la característica que define al objeto como entidad única dentro de un programa informático. Nos permite referirnos a él a lo largo del código fuente.

Podemos considerar a los **objetos** entidades independientes dentro del código, que utilizaremos para realizar acciones que intervengan en el curso de nuestro programa (Figura 2.1).

Por ejemplo, podemos crear el objeto *Impresora* (esta será su identidad), que admitirá métodos como imprimir un documento, abortar una impresión, vaciar la cola de documentos a imprimir, o suprimir un documento de la cola de impresión.

**Figura 2.1**  
Estructura básica (clase) de  
un objeto Coche.

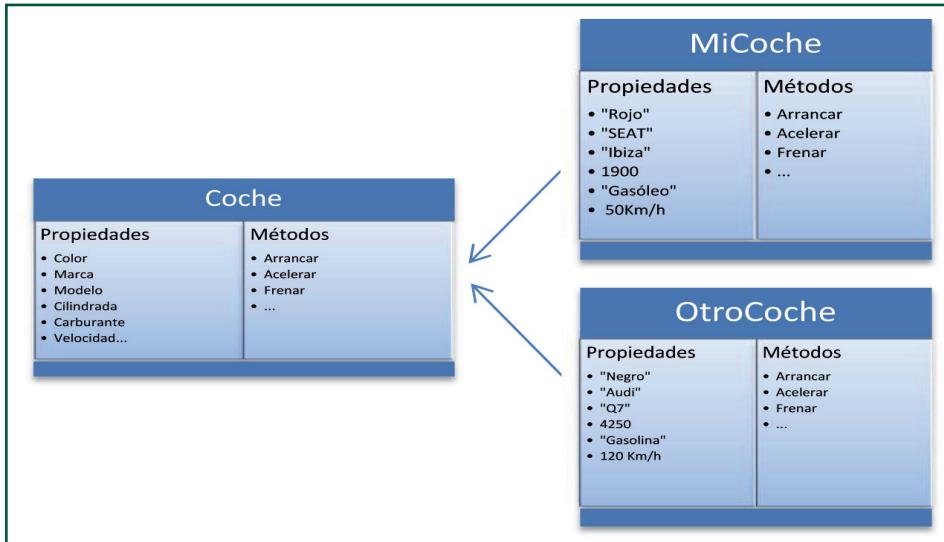
Coche	
<b>Propiedades</b> <ul style="list-style-type: none"> <li>• Color</li> <li>• Marca</li> <li>• Modelo</li> <li>• Cilindrada</li> <li>• Carburante</li> <li>• Velocidad...</li> </ul>	<b>Métodos</b> <ul style="list-style-type: none"> <li>• Arrancar</li> <li>• Acelerar</li> <li>• Frenar</li> <li>• ...</li> </ul>

## 2.2 Instanciación de objetos

Los objetos se definen utilizando **clases**. Una clase describe exactamente qué datos y métodos tendrá un determinado tipo de objeto.

La **instanciación** es el proceso a través del cual podremos generar un objeto en el código y usar sus funciones para que intervengan en la ejecución de nuestro programa.

Basta un simple operador **new** para generar (instanciar) una representación de un objeto a partir de su clase y empezar a operar con sus **métodos y propiedades**. De una misma clase pueden instanciarse tantos objetos como se considere oportuno (Figura 2.2.).

**Figura 2.2**

Ejemplo de instancias (objetos) de una clase

A continuación se expone cómo se instanciarían estos objetos en diferentes lenguajes, sin asignar los valores y asignándolos:

- Sin asignación de las propiedades básicas:

- Java: Coche miCoche = **new** Coche();

- C++:

```
Coche miCoche; o
Coche *otroCoche = new Coche0;
```

- Asignando las propiedades básicas en el momento de creación:

- Java: Coche miCoche = **new** Coche("Rojo", "SEAT", "Ibiza", ...);

- C++:

```
Coche miCoche("Rojo", "SEAT", "Ibiza", ...); o
Coche *otroCoche = new Coche("Negro", "Audi", "Q7", ...);
```

## 2.3 Utilización de métodos y propiedades

Una vez un objeto está instanciado en nuestro código, podremos acceder a sus **métodos y propiedades**. Veamos un ejemplo: tomemos un objeto que contiene una colección de métodos destinados a realizar operaciones matemáticas (Figura 2.3).

En este ejemplo en C++, primero instanciaremos la clase Aritmética:

*Aritmética calculadora;*

A continuación podremos usar sus métodos:

```
resultadoSuma = calculadora.suma(96, 35);
resultadoMultiplicacion = calculadora.multiplicar(96, 35);
resultadoDivision = calculadora.dividir(96, 35);
```

*calculadora* hace referencia a la **identidad**, el nombre, del objeto que estamos instanciando. La instrucción definida tras la identidad es el **método** que vamos a usar, y los valores escritos entre paréntesis son los parámetros que usará el método para calcular el resultado. El valor obtenido en cada operación será almacenado en el dato correspondientemente definido.

**Figura 2.3**

Estructura de un objeto definido en lenguaje de programación C++ que almacena una colección de métodos para realizar operaciones matemáticas.

```
1  class Aritmetica {
2      public:
3          inline int sumar (int a, int b) const
4          {
5              return a + b;
6          }
7          inline int restar (int a, int b) const
8          {
9              return a - b;
10         }
11         inline float multiplicar (int a, int b) const
12         {
13             return a * b
14         }
15     }
```

En cuanto a las propiedades, se accede con la notación *objeto.propiedad*. Por ejemplo, la clase Coche podríamos forzar el valor de su propiedad "velocidad" mediante la siguiente sentencia:

```
miCoche.velocidad=50;
```

En caso de C++, si se hubiera creado mediante un puntero, se haría de la siguiente forma:

```
otroCoche->velocidad=100.
```

La forma *objeto.propiedad* es la usada siempre por Java.

## Recuerda

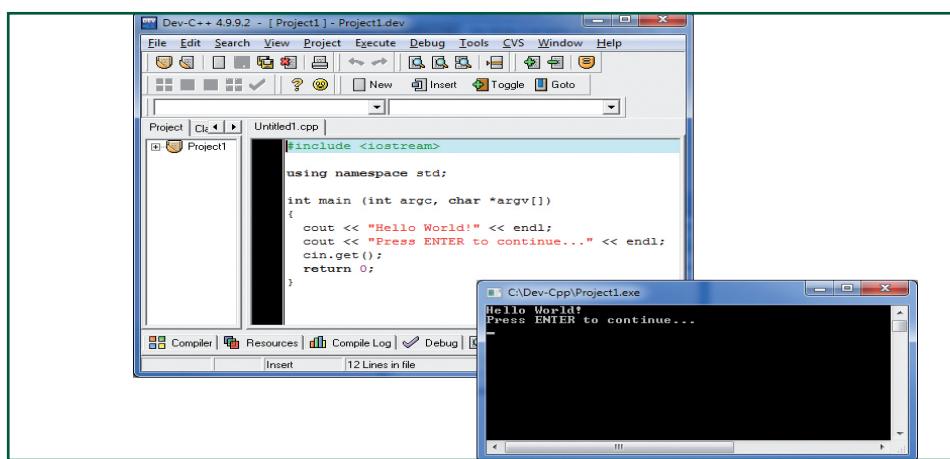
**La creación de objetos ayuda a separar las funciones que usamos, de forma puntual o momentánea, del código principal del programa.**

También en C++ se accedería de una de las siguientes formas, según fuera vía un objeto o un puntero:

```
miCoche.acelerar();
otroCoche->frenar();
```

## 2.4 Programación de la consola: entrada y salida de información

En programación, las **entradas** y las **salidas (E/S o I/O)** del original en inglés *input/output*) hacen referencia a la colección de señales enviadas y recibidas a lo largo de un programa informático. Mientras que las entradas son señales que transmiten información hacia la aplicación, las salidas son las señales de respuesta. Dicha información de salida puede ser visualizada por el usuario a través de interfaces determinadas como la **consola** (Figura 2.4). La entrada de información se producirá en modo consola por el teclado y la salida mediante una ventana modo texto. Las **consolas** son herramientas de control que permiten la comunicación entre el usuario y el equipo a lo largo de la ejecución de un programa. En C++ se utilizan los objetos `cin` y `cout`, y en Java los objetos `System.in` y `System.out`.



### Para saber más

**La información mostrada por la consola de programación no tiene por qué ser la representación final de nuestra aplicación, pero puede ayudarnos a controlar su buen funcionamiento durante el desarrollo; e incluso nos permite modificar aspectos del programa en tiempo de ejecución.**

**Figura 2.4**

Información de entrada y salida mostrada por la consola.

## 2.5 Utilización de métodos estáticos

### 2.5.1 Propiedades estáticas

Cuando instanciamos un objeto, definimos una representación de él en el código del programa. Todas las instancias de un objeto tendrán en común los diferentes métodos y propiedades definidas en su estructura. Al mismo tiempo, podemos encontrarnos en casos en los que necesitemos que alguno de los métodos o propiedades de un

objeto sea propio de la clase a la que pertenece y no del objeto instanciado. Para ello deberemos usar la etiqueta **static**, que lo definirá como **estático**. Cuando definimos una característica de un objeto como estática, las funciones que realiza se establecen a nivel global de la clase a la que pertenece y no a nivel local de su instancia.

Por ejemplo, tenemos la clase **Alumno** (Figura 2.5) con algunas de las características que representan a una persona del mundo real: nombre, edad, sexo, etc. Supongamos que necesitamos saber cuántas instancias de este objeto hay definidas a lo largo de nuestro código. Mientras la propiedad **nombre** pertenecería a cada uno de los objetos instanciados (un nombre para cada alumno), podremos crear otra propiedad de nombre **numAlumnos**, por ejemplo, y definirla como estática para que almacene el número de instancias del objeto que existen en el código.

**Figura 2.5**

Ejemplo de la clase Alumno con una variable estática en C++ (izquierda) y Java (derecha).

```

1  class Alumno {           public class Alumno {
2      char nombre[100];     String nombre;
3      int edad;           int edad;
4      char sexo; // 'v' p 'm'
5      static int numAlumnos; static int numAlumnos=0;
6
7      Alumno(){           Alumno(){
8          numAlumnos++;   numAlumnos++;
9      }                   }
10 };                     }
11 int Alumno::numAlumnos = 0; }
```

### 2.5.2 Métodos estáticos

Muchas clases, como Math (para cálculos matemáticos), String (manipulación de Strings) o las mismas clases *wraps*, que emulan los tipos base, como son Integer, Double o Char, ofrecen métodos estáticos. Estos métodos son genéricos y, por lo tanto, no es necesario instanciar objetos de la clase para utilizarlos. Se usan directamente con el nombre de la clase y el método:

```

double raiz= Math.sqrt(16);
double cuadrado=Math.pow(x,2);
System.out.println(" 25 en binario es " + Integer.toBinaryString(25));
```

## 2.6 Parámetros y valores devueltos

Denominamos **parámetros** a los datos usados por un método para realizar las acciones para las que ha sido definido. La recepción de estos datos le permitirá devolver un valor resultante que podrá ser almacenado en una variable o utilizado en una expresión. Tomemos como ejemplo un objeto que realiza el cálculo del área de

una circunferencia. Una vez instanciado el objeto en el código, podremos generar su método *areaCirculo* y enviarle los parámetros necesarios (en este caso, el radio) con el fin de obtener el resultado deseado.

La sintaxis para crear un objeto en Java y usar el método de manera adecuada sería la siguiente:

```
CalculoArea calculoArea = new CalculoArea();
float area = calculoArea.areaCirculo(10);
```

## Recuerda

**La instanciaión de objetos nos permite crear los objetos para así poder trabajar con sus métodos y propiedades.**

Debemos señalar que los parámetros se envían a través de variables definidas entre paréntesis, que el método recoge y asigna adecuadamente para la obtención del resultado. La instrucción **return** devolverá al código principal el valor del área resultante y quedará almacenado en la variable definida como *area*. Los parámetros pueden ser pasados a un método de dos maneras diferentes:

- **Por valor.** El parámetro pasado es un valor literal o el valor de una variable. El valor original de la variable en ningún caso se verá alterado por el método.
- **Por referencia.** El parámetro pasado al método es una variable. El valor de esta variable puede ser modificado desde el método.

En lenguaje C++, si no se indica lo contrario un array se pasa por referencia y una variable se envía por valor, como en el siguiente ejemplo:

```
tipo Metodo(tipo VariablePasadaPorValor simple) {
    // instrucciones del método
}
```

Para pasar una variable por referencia, a no ser que sea un array, que siempre y por defecto se pasa por referencia, sólo se necesita poner el carácter "&" justo después del nombre de la variable:

```
tipo Metodo(tipo &VariablePasadaPorReferencia) {
    // instrucciones del método
}
```

En Java, no hay una simbología especial para indicar el tipo de parámetro, es decir, si es por valor o por referencia, sólo las normas propias del lenguaje:

1. Los objetos se pasan por referencia.
2. Las referencias a objetos y los tipos elementales por valor.

Puedes observar que un método se define de manera parecida a una variable, indicando primero el tipo de datos que devuelve.

## 2.7 Librerías de objetos

Una librería de objetos es un fichero que contiene objetos útiles para diferentes programas. Estos objetos pueden ser **importados** por nuestros programas, ahorrándonos así el trabajo de tener que crearlos desde cero cada vez que los necesitemos.

Cuando se crea una librería de objetos se piensa en la resolución de problemas específicos. Así, por ejemplo, podemos tener librerías para resolver problemas matemáticos complejos, para controlar un robot industrial, para dibujar gráficos tridimensionales en la pantalla, o cualquier otra cosa que se nos ocurra.

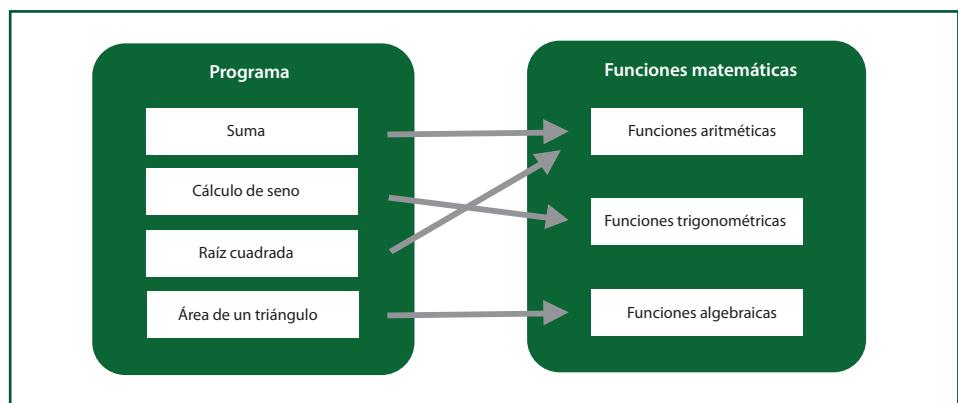
Los lenguajes de programación suelen incluir múltiples librerías para propósitos más o menos generales que no están contemplados en el lenguaje estándar.

Podemos hacer nuestras propias librerías, que nos servirán para reutilizar objetos interesantes y para mantener los objetos de un programa grande bien organizados.

También encontramos librerías comerciales, que nos son ofrecidas para resolver problemas complejos y específicos.

Siguiendo los ejemplos expuestos hasta ahora, podríamos definir una librería que englobara distintas funciones matemáticas según su tipo (Figura 2.6).

**Figura 2.6**  
Relación entre el programa principal y una librería de objetos de funciones matemáticas.



## 2.8 Constructores

Definimos como **constructor** la función que se ejecutará de forma inmediata al instanciar un objeto en el código y realizará las acciones que tenga definidas. La finalidad del constructor es inicializar el objeto y permitir que éste tenga una identidad dentro del programa. El único requisito que debemos tener en cuenta es que su nombre debe ser exactamente el mismo que el de la clase de objeto al que representa o, de otro modo, el programa dará error. Por norma general, el nombre de las clases y sus constructores se define con la primera letra de su nombre en mayúscula. Si el nombre del constructor incluye más de una palabra, ésta se definirá también con la primera letra en mayúscula y sin separación con la anterior.

## 2.9 Destrucción de objetos y liberación de memoria

La **destrucción de objetos** es uno de los aspectos más importantes a tener en cuenta cuando diseñamos un programa informático. Igual que las variables y todos los otros tipos de almacenamiento de datos, cuando definimos un objeto, éste pasa a ocupar una parte de la memoria de nuestro ordenador. La destrucción de un objeto implica que el objeto deja de existir en el programa y, por consiguiente, deja de estar almacenado en la memoria de nuestra máquina.

Cuando el código es sencillo, es posible que éste no sea un aspecto especialmente visible, lo que no significa que deje de ser importante; pero, cuando hablamos de programas complejos, resulta imprescindible. Toda la memoria que podamos ahorrar a la hora de ejecutar un programa, contribuirá a que éste funcione de una manera más ágil en el ordenador. Si no tenemos en cuenta la liberación de memoria durante la confección de nuestro código, corremos el riesgo de que el programa llegue a colapsarse en el momento de su ejecución. Para evitar esto, los lenguajes de programación incluyen ciertas formas que nos ayudarán, de manera sencilla, a liberar la memoria de aquellos datos que ya no utilizamos.

En Java, la destrucción de objetos se realiza automáticamente mediante el *garbage collector*, o recolector de basura, que efectúa la máquina virtual. Sin embargo, Java ofrece la posibilidad de implementar el método `finalize` de la clase del objeto, para ejecutar ciertas acciones antes de que se elimine por completo.

```
protected void finalize() {
    // Código a ejecutar antes de liberar el objeto
}
```

## Resumen

Los objetos son, sin duda, una de las técnicas de programación más importantes a tener en cuenta a la hora de diseñar y escribir nuestros programas. Las características de los objetos se denominan propiedades, que son variables de cualquier tipo de dato, incluso otros objetos. El comportamiento de los objetos se establece con los métodos, es decir, funciones contenidas en los propios objetos, con o sin parámetros de entrada y devolviendo un valor o no.

Las propiedades pueden ser utilizadas generalmente con el formato "nombre de objeto.nombre de propiedad".

Los métodos de los objetos, como se ha mencionado, son funciones contenidas en el propio objeto. Como con las propiedades, pueden ser llamados como "nombre de objeto.nombre de método(parámetros)".

El diseño o codificación de los objetos se realiza a través de lo que denominamos clase. Una clase no es más que un tipo de datos donde quedan definidos propiedades y métodos. Cuando creamos un objeto a partir de una clase, se dice que estamos instanciando el objeto, y en ese momento se ejecuta el método constructor de la clase. Si utilizamos un método o propiedad directamente desde la clase, sin instanciar un objeto, se habla de propiedades y métodos estáticos.

En Java, aparte de las palabras reservadas del lenguaje, se utilizan librerías de clases realizadas por terceros o por nosotros mismos. Las clases de una librería pueden agruparse en conjuntos denominados paquetes. Un ejemplo es la clase System, que contiene los objetos in y out que permiten leer y escribir datos en la consola, respectivamente.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. En programación, un objeto es una entidad provista de un conjunto de propiedades (métodos) y de comportamientos (datos) que reaccionan según determinados eventos.
2. La instanciación es el proceso a través del cual podremos generar un objeto en el código y usar sus funciones para que intervengan en la ejecución de nuestro programa.
3. Las consolas son herramientas de control que permiten la comunicación entre el usuario y el equipo a lo largo de la ejecución de un programa. En C++ se utilizan los objetos *cin* y *cout*, y en Java los objetos *System.in* y *System.out*.
4. Denominamos valores devueltos a los datos usados por un método para realizar las acciones para las que ha sido definido. La recepción de estos valores le permitirá devolver un dato resultante que podrá ser almacenado en una variable o utilizado en una expresión.
5. Una librería de objetos es un fichero que contiene objetos útiles para diferentes programas. Estos objetos pueden ser importados por nuestros programas.
6. La finalidad del constructor es inicializar el objeto y permitir que este tenga una identidad dentro del programa.
7. En Java, la destrucción de objetos se realiza automáticamente mediante el *garbage collector*, o recolector de basura, que efectúa la máquina virtual.

**Completa las siguientes afirmaciones:**

8. Basta un simple operador *new* para generar (instanciar) una representación de un objeto a partir de su \_\_\_\_\_ y empezar a operar con sus \_\_\_\_\_ y propiedades. De una misma clase pueden instanciarse tantos \_\_\_\_\_ como se considere oportuno.

9. La instrucción \_\_\_\_\_ devolverá al código principal el valor del área resultante y quedará almacenado en la \_\_\_\_\_ definida como \_\_\_\_\_.

10. Cuando creamos un objeto a partir de una clase, se dice que estamos \_\_\_\_\_ el objeto, y en ese momento se ejecuta el método \_\_\_\_\_ de la \_\_\_\_\_.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 3. USO DE ESTRUCTURAS DE CONTROL

Hasta ahora hemos visto cuáles son los elementos principales que intervienen en la programación y hemos aprendido cómo el uso de datos y objetos nos ayudan a organizar nuestro código. Sin embargo, para la resolución de problemas complejos, es preciso conocer cuáles son las sentencias o acciones que se ejecutan, y en qué momento se ejecutan. Las **estructuras de control**, o **sentencias de control**, controlan la secuencia o flujo de ejecución de las acciones que realiza nuestro programa. A lo largo de esta unidad, aprenderemos a controlar los sucesos que desencadenarán las acciones que definiremos en nuestro código durante la ejecución de nuestros programas. La función de las **sentencias de control** es la de dirigir el flujo de las acciones a lo largo de un programa informático con el objetivo de que éste reaccione de un modo u otro según la situación. Hay estructuras de varios tipos (selección, repetición, salto y excepción). Todas ellas gozan de características y utilidades distintas. Veamos pues, cuáles son y en qué situaciones deberemos usar unas u otras.

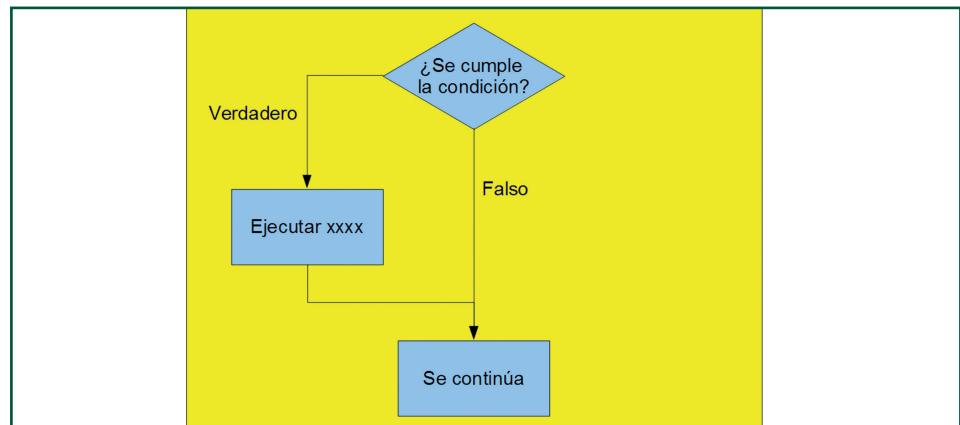
### 3.1 Estructuras de selección

Las **estructuras de selección** definen diferentes comportamientos para nuestro programa basándose en la evaluación de sentencias o premisas (condiciones) (Figura 3.1). Existen tres tipos:

- **Selección simple.** Su comportamiento se basa en la evaluación de una condición y reacción en consecuencia. Si la premisa se cumple, se desencadenan las acciones definidas; de lo contrario, el programa sigue su curso. La instrucción usada es el condicional *if* (*si*).
- **Selección doble.** Amplía la funcionalidad de las estructuras de selección simple añadiendo un camino alternativo si la premisa evaluada no se cumple. Las instrucciones que las definen son *if / else* (*si / si no*).
- **Selección múltiple.** Se basan en la evaluación de colecciones de premisas. El programa elegirá el comportamiento adecuado según sea la premisa que se cumpla. La instrucción que las define se denomina *switch* (*cambia a/selección*), y los diferentes comportamientos vienen determinados según las sentencias *case* (*caso*).

Para ejemplificar su uso, podríamos proponer la siguiente situación: imaginemos la página web de una marca de bebidas alcohólicas en la que el acceso está vetado a los menores de 18 años. Cuando un usuario intenta acceder, se le pide la fecha de nacimiento para realizar la comprobación de su edad según el año

en curso. Una vez el programa define que el usuario es mayor de 18 años, se le permite el acceso al contenido de la web; de otro modo, seguirá vetado.

**Figura 3.1**

Esquema conceptual de una estructura de selección simple.

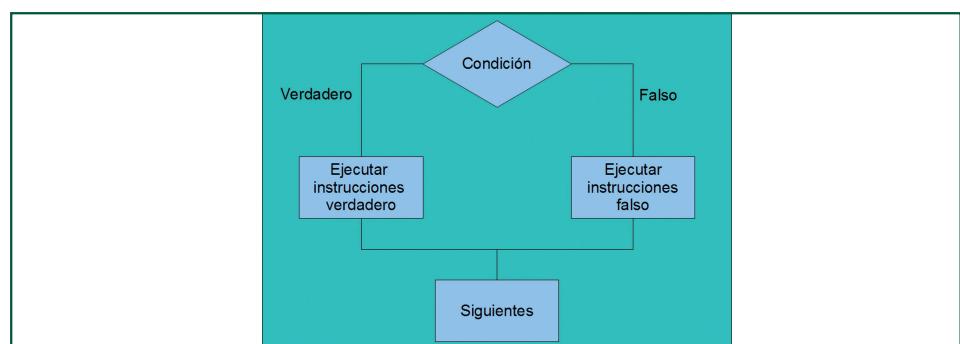
### 3.1.1 Estructuras de selección en Java

**La sentencia *if*, o *if-else*:**

```

if ( <expresión lógica> ) {
    <lo que sea>;
}
[ else {
    <lo que sea>;
} ]
  
```

La expresión lógica tiene que producir un resultado lógico, es decir, *true* o *false* (Figura 3.2). La sentencia puede ser simple, acabada en punto y coma, o puede ser compuesta, siempre encerrada por llaves. Aunque sólo sea necesario el uso de las llaves cuando haya más de una sentencia, se recomienda usarlas siempre para aumentar la legibilidad.

**Figura 3.2**

Esquema conceptual de una estructura de selección simple con doble bifurcación.

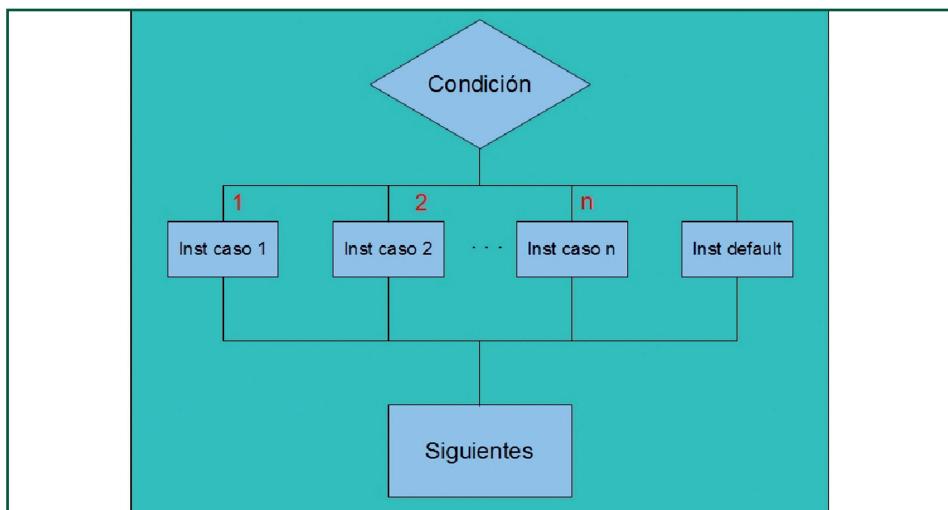
### La selección múltiple, sentencia *switch –case*:

```
switch( <expresión> ) {
    case valor1:
        <lo que sea>;
        break;
    case valor2:
        < lo que sea >;
        break;
    [default:
        < lo que sea >]
}
```

La expresión, que puede ser una variable, debe dar como resultado *int* o *char*.

Cada *case* acaba en un *break* y esto hace que, después de cada evaluación y ejecución, se vaya al final. No es obligatorio, pero se suele utilizar para que cada *case* sea exclusivo. Si no se pusiera, entraría en los demás *case*, aunque no tuviera ese valor.

Al final, si hay un *default*, el código de dentro se ejecutará cuando no se haya encontrado coincidencia anterior (Figura 3.3).



**Figura 3.3**

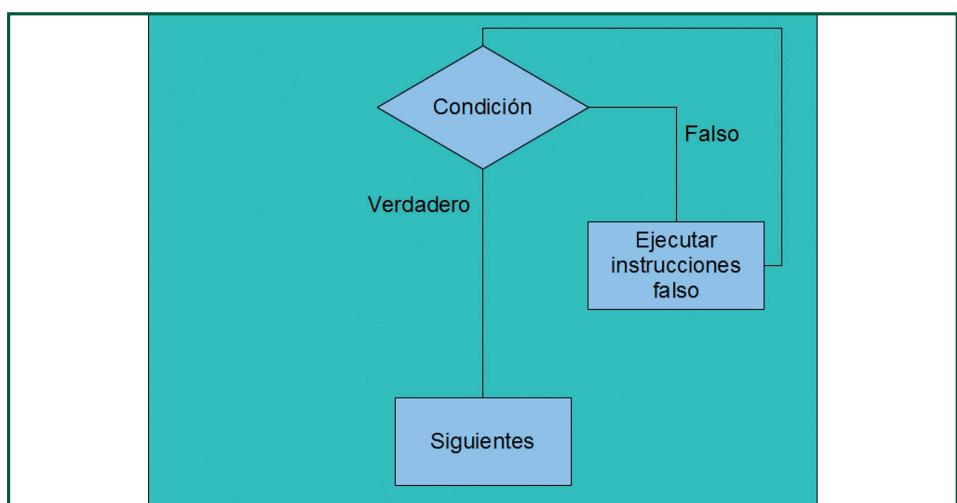
Esquema conceptual de una estructura de selección múltiple.

## 3.2 Estructuras de repetición

Las **estructuras de repetición** se basan en la reiteración de una acción definida mientras se cumpla una premisa expuesta. Existen de varios tipos:

- **for.** Determina un número máximo de repeticiones durante las que se ejecutará una acción definida. Cuando se alcanza el número de repeticiones indicado, el bucle acaba y pasa a la siguiente línea de código. Es muy útil cuando sabemos que se debe repetir algo "N" veces y nos va bien que un contador se vaya incrementando o decrementando.
- **while.** Su funcionamiento es parecido al de la sentencia *for*. El programa ejecuta una acción mientras se cumpla una condición definida. Para que el bucle se rompa, deberá existir en el interior de las acciones definidas una instrucción que convierta la condición en falsa. De otro modo el programa entraría en un bucle infinito y se bloquearía.
- **do while.** Muy similar a la sentencia *while*, la diferencia es que la comprobación de la premisa se realiza posteriormente a las acciones definidas, es decir, las acciones se ejecutarán como mínimo una vez.

Todas las estructuras tienen una utilidad y función según la situación; sin embargo, son las estructuras *for* y *while* las más usadas de todas ellas (Figura 3.4).



**Figura 3.4**

Esquema conceptual de una estructura de repetición.

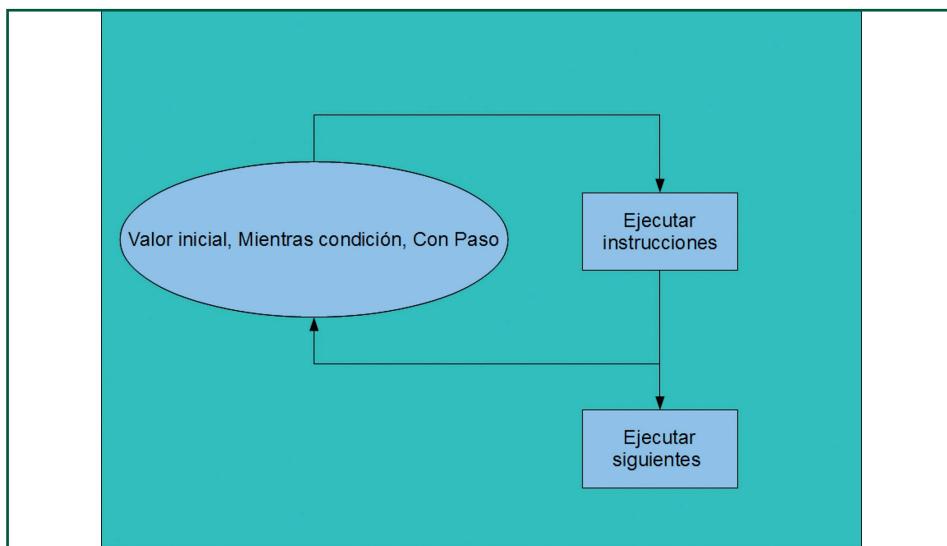
### 3.2.1 Estructuras de repetición en Java

#### La sentencia *for*:

```
for ( < inicialización>; < finalización>; < iteración> ) {  
    < lo que sea>;  
}
```

La sentencia `for` ejecuta un grupo de instrucciones repetidamente con las siguientes restricciones (separadas por punto y coma) (Figura 3.5):

- **Inicialización.** Una expresión que inicializa la variable o variables que se utilizan en la iteración. Por ejemplo, `i = 0`.
- **Finalización.** Mientras se cumplan las condiciones que se incluyen, se irán ejecutando repetidamente las instrucciones contenidas en el `for`. Por ejemplo, `i < 5`.
- **Iteración.** Incremento o decremento de la variable o variables que a cada iteración se va a actualizar. Por ejemplo, `i++` hará que a cada vuelta se incremente en 1 la variable `i`, pero también podría ser `i--`, o `i = i + 2`.



**Figura 3.5**  
Esquema conceptual de una estructura `for`.

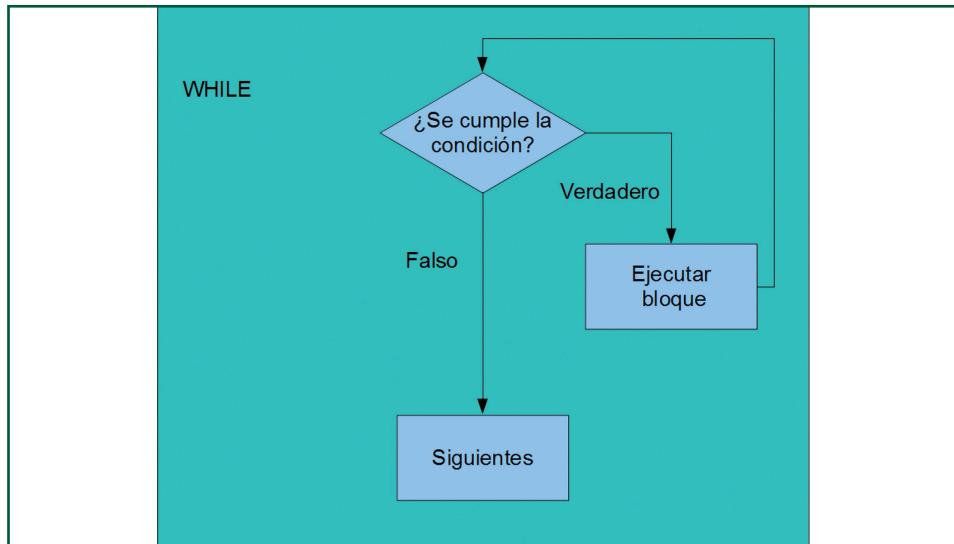
### La sentencia `while`:

```

[ <inicialización>; ]
while ( <expresión condicional> ) {
    <lo que sea>;
    <iteración>;
}
  
```

Esta estructura ejecuta repetidamente las sentencias que encierra mientras la expresión condicional sea verdadera; por lo que, si es inicialmente falsa, no se ejecuta. Por eso si es necesario que al menos entre una vez se inicializa, y en las sentencias de iteración es donde se cambia algo para que en algún momento no se cumpla la expresión condicional (Figura 3.6).

**Figura 3.6**  
Esquema conceptual de una estructura *while*.



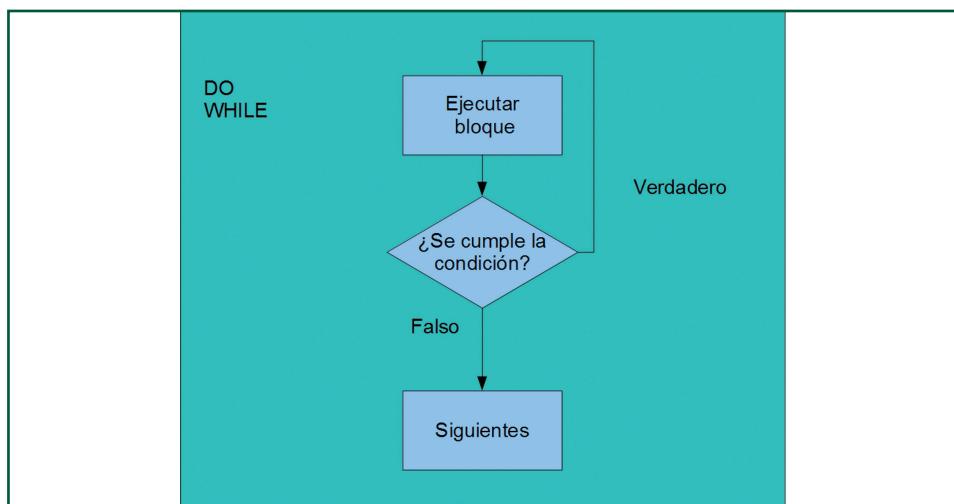
### La sentencia *do - while*:

```

[<inicialización>;]
do {
    <lo que sea>;
    [ <iteración>; ]
} while ( <expresión condicional> );
  
```

Esta estructura ejecuta repetidamente las sentencias que encierra mientras la expresión condicional sea verdadera. Sin embargo, como la condición está al final, a diferencia del *while*, la primera vez se ejecutará siempre (Figura 3.7).

**Figura 3.7**  
Esquema conceptual de una estructura *do-while*.

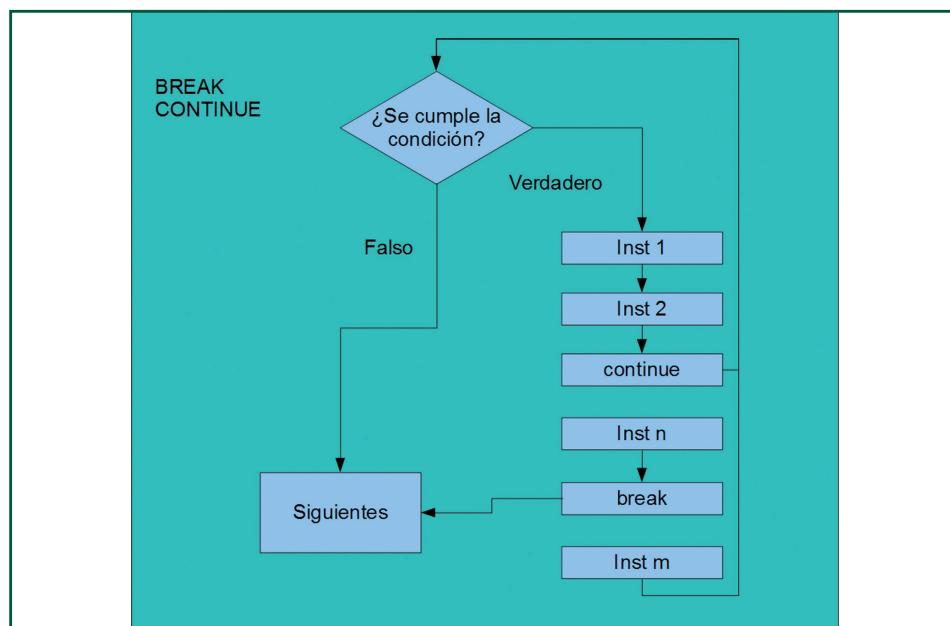


### 3.3 Estructuras de salto

Las **estructuras de salto** sirven para interrumpir el flujo normal del programa y desviarlo hacia un punto diferente (Figura 3.8). Podemos destacar tres tipos:

- ***break***. Provoca una salida brusca de un bucle *for*, *while* o *do-while*. También en la sentencia *switch*, permite que los case sean exclusivos. Tras ella, continúa la ejecución del resto del programa.
- ***continue***. Interrumpe la ejecución normal de un bucle, ir directamente a la siguiente repetición, pero sin finalizar el bucle que continua su proceso.
- ***return***. Sólo se permite su uso dentro de funciones o bloques de instrucciones. Provoca la salida inmediata del bloque y retorna al punto desde donde había sido activado, devolviendo, si es necesario, un valor.

Es importante no usar estas acciones más que en las situaciones definidas. Cualquier otro uso forzado podría provocar el colapso del programa.



**Figura 3.8**  
Esquema conceptual de las diferentes estructuras de salto y sus usos permitidos.

### 3.4 Control de excepciones

El **control de excepciones** es una estructura que se utiliza para evitar que un programa provoque un error y se cierre cuando intenta ejecutar un código no válido o que no pueda interpretar. Una estructura de control de excepciones se compone de dos grandes bloques:

## Para saber más

**Las estructuras de control son combinables entre sí y, aunque no es aconsejable abusar de ello, si la ocasión lo requiere, pueden incluso anidarse las unas con las otras para crear nuevas estructuras.**

- **Bloque try.** se definen las sentencias a ejecutar y que pueden producir un error. En caso de éxito, el programa sigue su flujo normal, en caso contrario pasa a la siguiente instrucción *catch*.
- **Bloque catch.** Bloque en el que estarán las instrucciones que gestionarán el error generado.

Como ejemplo de uso para una estructura de control de excepciones podríamos tomar una función cuyos parámetros serán usados para realizar una división. Si al parámetro definido como divisor se le asignara el valor 0, el programa podría colapsarse al no poder realizar la operación. Gracias al control de excepciones (Figura 3.9) el programa podrá determinar que ha ocurrido un error, actuar en consecuencia, advertir de lo ocurrido al usuario y seguir su curso normal.

```

1 int division(int x, int y)
2 {
3     try
4     {
5         return (x/y);
6     }
7     catch (error e)
8     {
9         throw new error ("parámetros de división no evaluables.", e);
10        return 0;
11    }
12 }
```

Figura 3.9

Esquema conceptual del proceso de control de excepciones en una función en C++.

### 3.4.1 Excepciones en Java

```

try {
    <lo que sea>;
} catch(Exception e) {
    <lo que sea>;
}
```

## Recuerda

**Las estructuras de salto están diseñadas para ser introducidas en ciertos puntos concretos del código en los que se requieren. Su uso fuera de esos puntos no es aconsejable, salvo que no se halle una mejor solución.**

Al ejecutar las sentencias incluidas en el *try*, si se produce un error, Java creará un objeto de la clase *Exception* o de una de sus heredadas. Este objeto será pasado al *catch* y se ejecutarán las sentencias que se incluyen en el mismo.

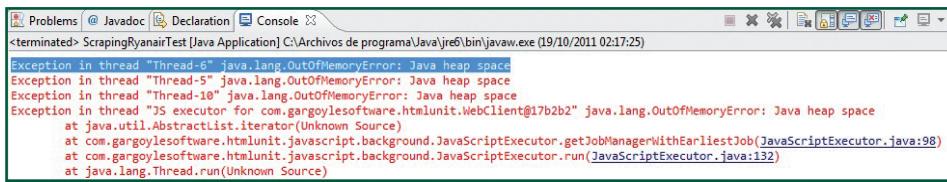
Se pueden incluir varias cláusulas *catch* con distintos tipos de excepciones. Siempre se deben poner de mayor a menor detalle. Es decir, como la clase *Exception* es de donde heredan todas, ésta siempre irá la última y, por lo tanto, el grupo de sentencias de su *catch* siempre se ejecutará si no se ha cazado en una excepción más particular de un *catch* anterior del mismo *try–catch*.

### 3.5 Prueba y depuración

Una vez hemos acabado de escribir el código de nuestro programa llega el momento de someterlo al proceso de **prueba y depuración**. La finalidad de las pruebas es definir si las acciones que realiza el programa durante su ejecución dan como resultado la resolución del problema para el que ha sido diseñado. Si los resultados obtenidos no son satisfactorios deberemos revisar nuestro código, detectar en qué puntos se producen errores y depurarlos. La prueba y la depuración son procesos que deben hacerse tanto a lo largo de la escritura del programa, para comprobar la funcionalidad de pequeños bloques de código, como al finalizar la aplicación para comprobar su funcionalidad con respecto a los usuarios. Es en este proceso donde toma mayor importancia la consola de errores (Figura 3.10), ya que nos ayudará a detectar los puntos donde se producen errores y conocer de qué tipo son.



Para ampliar este tema, puedes ver el videotutorial *Creación de aplicaciones con Netbeans*, que encontrarás en el Campus online.



```

Problems @ Javadoc Declaration Console 
<terminated> ScrapingRyanairTest [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe (19/10/2011 02:17:25)
Exception in thread "Thread-6" java.lang.OutOfMemoryError: Java heap space
Exception in thread "Thread-5" java.lang.OutOfMemoryError: Java heap space
Exception in thread "Thread-10" java.lang.OutOfMemoryError: Java heap space
Exception in thread "JS executor for com.gargoylesoftware.htmlunit.WebClient@17b2b2" java.lang.OutOfMemoryError: Java heap space
    at java.util.AbstractList.iterator(Unknown Source)
    at com.gargoylesoftware.htmlunit.javascript.background.JavaScriptExecutor.getJobManagerWithEarliestJob(JavaScriptExecutor.java:98)
    at com.gargoylesoftware.htmlunit.javascript.background.JavaScriptExecutor.run(JavaScriptExecutor.java:132)
    at java.lang.Thread.run(Unknown Source)

```

**Figura 3.10**

Listado de errores mostrados por la consola durante la ejecución de un programa.

### 3.6 Documentación

La documentación es la guía o comunicación escrita que ayuda a comprender el uso de un programa y facilita sus futuras modificaciones (mantenimiento). Debe recoger todos los elementos y material creado en las diferentes fases del desarrollo, además de las normas de instalación y recomendaciones para su ejecución. La documentación se puede dividir en tres partes:

- **Documentación interna.** Son los comentarios que se añaden al código fuente para clarificarlo. Es muy importante también que se utilicen nombres para identificar las variables, propiedades, métodos... que recuerden su utilidad. Esto, junto con los comentarios, hace que el código fuente, ya de por sí, esté documentado.
- **Documentación externa.** Es el documento que recoge todo el material creado y empleado en las diferentes fases del desarrollo del programa. Las bases de los puntos que debe incluir son:

- Descripción del problema
- Explicación de la solución
- Autores
- Algoritmo (diagrama de flujo o pseudocódigo)

## Recuerda

**La documentación es uno de los procesos que suelen pasarse por alto; sin embargo, supone el complemento perfecto para un producto final usable y estable.**

- Código Fuente (programa)
- Limitaciones del programa (funciones que puede y no puede realizar)

- **Manual del usuario.** Es el documento que describe paso desde los requerimientos del sistema y la forma de instalación, hasta llegar a la descripción detallada de su funcionamiento. Su finalidad es que los usuarios entiendan el modo correcto de usarlo para obtener los resultados deseados.

## Resumen

Como cualquier lenguaje de programación, Java debe implementar las estructuras de control que permiten cambiar la ejecución secuencial del programa y, en definitiva, nos ayudan al implementar cualquier tipo de algoritmo.

Las estructura de control de selección, if – else , ejecuta las instrucciones contenidas en el if siempre que se cumpla la condición; en otro caso, ejecutará las del else, siempre que se haya explicitado esta palabra clave.

La estructura de control switch – case evalúa el valor de una variable y ejecuta las instrucciones que se encuentran en el case correspondiente al valor.

Las estructuras de repetición son estructuras de control que ejecutan un conjunto de instrucciones repetitivamente. En la estructura while, se ejecuta el conjunto mientras se cumpla la condición. La estructura do – while, ejecuta primero el conjunto de instrucciones y después evalúa, repitiendo el proceso mientras se cumpla la condición. La estructura for(inicialización;condiciones;incremento) ejecuta el conjunto inicializando unas variables, mientras se cumplan las condiciones e incrementando o decrementando las variables de la parte del incremento.

En Java, se puede prever un error y ejecutar unas instrucciones cuando ocurre. Este proceso se denomina control de excepciones, y se realiza mediante la estructura try - catch. Cuando las instrucciones dentro del try tienen un error se ejecutarán las instrucciones contenidas en el catch.

Los entornos de programación integrados (IDE, del acrónimo en inglés) tienen utilidades para depurar los programas, es decir, seguirlos paso a paso para encontrar los posibles orígenes de un mal funcionamiento.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Se pueden incluir varias cláusulas *catch* con distintos tipos de excepciones, aunque siempre se deben poner de menor a mayor detalle.
2. La función de las sentencias de control es la de dirigir el flujo de las acciones a lo largo de un programa informático, con el objetivo de que este reaccione de un modo u otro según la situación.
3. La estructura de salto *return* provoca la salida inmediata del bloque y retorna al punto desde donde había sido activado, devolviendo, si es necesario, un valor.
4. La estructura de repetición *for* determina un número mínimo de repeticiones durante las que se ejecutará una acción definida. Cuando se alcanza el número de repeticiones indicado, el bucle acaba y ya no pasa a la siguiente línea de código.
5. Las estructuras de control pueden ser de cuatro tipos: selección, repetición, salto y excepción.
6. Las estructuras de control son combinables entre sí y, aunque no es aconsejable abusar de ello, si la ocasión lo requiere, pueden incluso anidarse las unas con las otras para crear nuevas estructuras.
7. La prueba y la depuración son procesos que deben hacerse solo a lo largo de la escritura del programa, para comprobar la funcionalidad de pequeños bloques de código, y nunca al finalizar la aplicación.

**Completa las siguientes afirmaciones:**

8. Existen tres tipos de estructuras de salto, y son las siguientes: \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
9. La estructura de control de selección doble amplía la funcionalidad de las estructuras de \_\_\_\_\_ simple añadiendo un camino alternativo si la premisa evaluada no se cumple. Las instrucciones que las definen son \_\_\_\_\_ /\_\_\_\_\_.

10. El control de excepciones es una estructura que se utiliza para evitar que un programa provoque un error y se cierre cuando intenta ejecutar un \_\_\_\_\_ no válido o que no pueda interpretar. Una estructura de control de excepciones se compone de \_\_\_\_\_ grandes bloques.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 4. DESARROLLO DE CLASES

En programación, el concepto de **clase** (*class*) está directamente relacionado con el concepto de **objeto** (*object*).

Se define una clase como el concepto bajo el que se agrupan objetos de un mismo tipo.

Si tomamos un objeto como la representación abstracta de un concepto real, una clase sería aquello que define de qué tipo de objeto se trata. Si tomamos el ejemplo de un coche, como concepto tangible, podríamos representarlo según unas características propias: su color, la velocidad máxima que puede alcanzar, su cubicaje, etc. Sin embargo, si tomamos el ejemplo de una motocicleta, veremos que, salvando las diferencias, muchas de las características que podemos asignarle coinciden con las de un coche. Ambos poseen velocidad máxima y cubicaje, color, marca, ruedas, en definitiva, podríamos afirmar que ambos pertenecen a un mismo grupo: el de los vehículos.

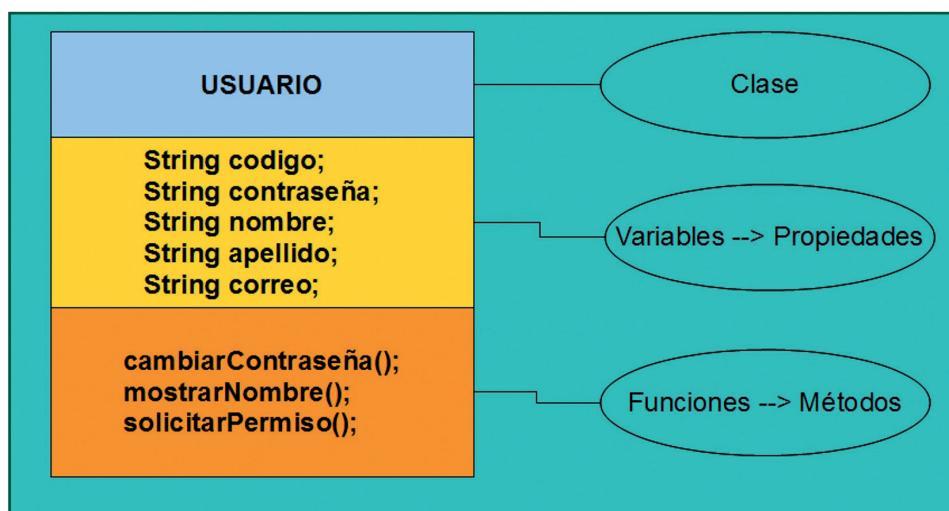
Una clase es, pues, el concepto bajo el cual se engloban los objetos que comparten una amplia serie de características comunes. La clase vehículo sería el grupo principal de toda una serie de objetos que constituyen diversas subclases, en ese caso: coches, motocicletas, camiones, furgonetas, etc. Como objeto principal, la clase vehículo contendría en sí misma todas las características compartidas por aquellos objetos que derivan de ella. Y, en tanto que objetos derivados, un coche o una motocicleta tendrían, además de sus características comunes, las suyas propias.

Las clases son uno de los aspectos clave de la **Programación Orientada a Objetos (POO)**. Sobre ellas se basa la estructura y funcionamiento de los programas informáticos. Si dominamos y comprendemos su filosofía, podremos diseñar programas que posean una estructura lógica firme, que contribuirán a un mejor funcionamiento de la aplicación final, así como a una mayor facilidad en el manejo y mantenimiento.

### 4.1 Estructura y miembros de una clase

En términos generales, una clase se estructura exactamente igual que un **objeto** (Figura 4.1). Recordando la estructura de un objeto, hay un primer bloque para los atributos, llamados aquí **variables miembro**, y un segundo bloque de métodos o acciones. Las variables miembro y los **métodos** definidos en una clase serán compartidos o heredados por todos los objetos que de ella deriven. A las **variables miembro** se puede otorgar la característica *private* (privado), que impide que

sean modificadas libremente desde un punto externo a la clase. Este concepto responde a un principio de la **POO** llamado **ocultación**, que, indica que para modificar o acceder a una variable propia de una clase, es preciso utilizar un paso intermedio llamado **mensaje**. Los mensajes se usan tanto para acceder y modificar las variables miembro de una clase, como para comunicar clases entre sí. Según el lenguaje de programación, existen varios modos para comunicar clases, pero el más utilizado es a través de los **métodos** especiales (*getters* y *setters*, Figura 4.2) que, mediante instrucciones concretas, permiten modificar y obtener los valores de las variables miembro de una clase. En los siguientes apartados de esta unidad veremos con más detalles algunos de estos conceptos.

**Figura 4.1**

Esquema de la estructura básica de una clase y sus diferentes bloques.

```

1  public class Persona {
2      protected String nombre;
3      protected String apellido1;
4      protected String apellido2;
5
6      private int edad;
7
8      public void setEdad(int edad){
9          if(edad > 0) {
10              this.edad = edad;
11          }
12      }
13      public int getEdad(){
14          return (this.edad);
15      }
16  }
17

```

**Figura 4.2**

Ejemplos de *setter* y *getter*.

### 4.1.1 Sintaxis de una clase en Java

```
class Persona{
    //propiedades
    ...
    //métodos
    ...
}
```

La definición completa de una clase en Java es:

[public] [final | abstract] class Clase [extends ClaseMadre] [implements Interfase1  
[, Interfase2]...]

Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos pues, con la declaración simple.

### 4.2 Creación de atributos

Cuando hablamos de **atributos** de una clase, nos referimos a todo aquel dato destinado a ser compartido por todos los objetos que de ella deriven. Los **atributos** pueden ser de tres tipos:

- **Públicos (public)**. Son aquellos a los que puede accederse o modificarse directamente tanto a nivel interno como externo de la clase.
- **Privados (private)**. Denominados también **variables miembro**, son aquellos que sólo pueden ser modificados directamente a nivel interno de la clase. Para modificar o acceder a un atributo privado desde un punto externo a la clase a la que pertenece, deberemos utilizar un proceso llamado **mensaje**, el cual establece unos métodos (funciones) específicos para tales efectos.
- **Protegidos (protected)**. Similar al anterior, pero accesible desde las clases derivadas.

### 4.2.1 Sintaxis de atributos en Java

Los atributos de una clase en Java son las propiedades del mismo.

```
private | public class Persona{
    //propiedades
    int i=0;
    char letra ='a';
    //métodos
    ...
}
```

Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos pues, con la declaración simple. Sin embargo, hay que recordar que una propiedad estática (*static*) es aquella que es única a nivel de clase, no de objeto.

### 4.3 Creación de métodos

Los métodos de una clase hacen referencia a las acciones (funciones) comunes a todos los objetos que de ella se deriven. Igual que sus atributos, pueden ser de tipo público (*public*), privado (*private*), o protegido (*protected*):

- **Métodos públicos (*public*).** Un método público puede ser accedido directamente tanto a nivel interno como externo de la clase. Por norma general se utilizan para impartir órdenes a una clase con el fin de que active una de sus funciones.
- **Métodos privados (*private*).** Sólo pueden ser llamados a nivel interno de la clase. Hacen referencia a funciones propias de un objeto, algo que éste no tiene por qué compartir con otras clases u objetos derivados.
- **Métodos protegidos (*protected*).** Similar al anterior, pero accesible desde las clases derivadas

Tal y como hemos visto en el apartado anterior, existen también métodos públicos específicos para acceder a los atributos privados de una clase. Son los llamados *getters* y *setters* (Figura 4.3):

```

1 static int nPersonas=0;
2 public Persona(){
3     this.nombre="Desconocido";
4     this.apellido1="Desconocido";
5     nPersonas++;
6 }
7 public Persona(String nom, String ap1){
8     this.nombre = nom;
9     this.apellido1=ap1;
10    nPersonas++;
11 }
12 public void finalize(){
13     nPersonas--;
14 }
15

```

**Figura 4.3**

Ejemplo en Java de métodos públicos *getter* y *setter* para el acceso/modificación de propiedades privadas.

## Recuerda

**Los métodos definen las funciones que serán capaces de desempeñar nuestras clases a lo largo de la ejecución del programa. Según la estructura con la que organicemos nuestras clases, ésta nos permitirá evitar la repetición de métodos para todas aquellas clases con características comunes.**

- **Getters.** Sirven para conocer el valor de un atributo en concreto. Es un método de sólo lectura que se limita a devolver el valor de un único dato en concreto y, en ningún caso, podrá modificarlo.
- **Setters.** Sirven para asignar valor a un atributo, bien para modificarlo, bien para iniciarla si éste estaba vacío.

### 4.3.1 Sintaxis de métodos en Java

Los métodos de una clase en Java son acciones que ofrecerá el objeto. Los modificadores se irán explicando a lo largo del curso en las situaciones concretas donde se utilicen. Nos quedamos, pues, con la declaración simple.

## 4.4 Creación de constructores

Igual que en la estructura de un objeto, llamamos **constructor** a la función especial de una clase. La finalidad del constructor es inicializar la clase y permitir que esta tenga una identidad única dentro del programa. Los únicos requisitos que hay que tener en cuenta es que se deberá definir como público (*public*) y su nombre deberá ser exactamente el mismo que el nombre de la clase a la que representa (Figura 4.4).

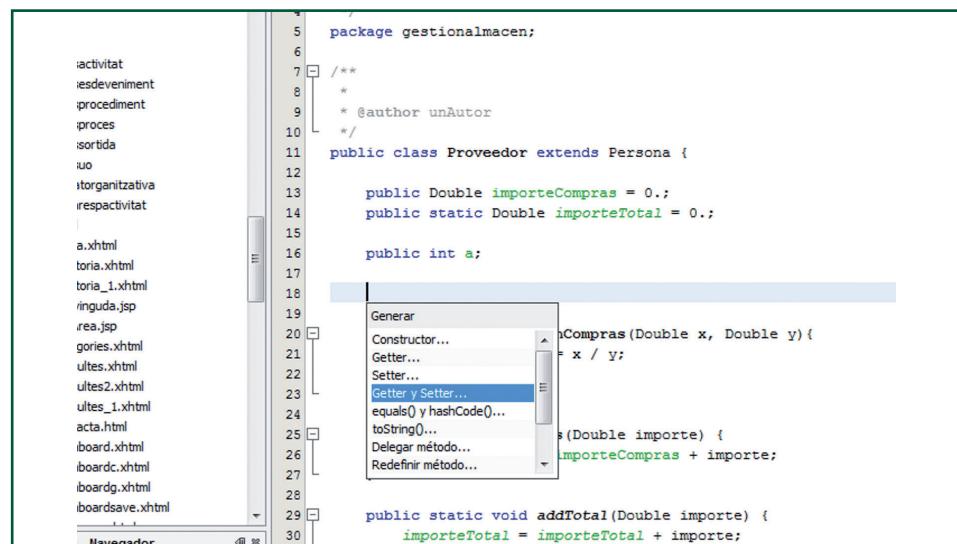


Figura 4.4

Muestra de ayuda del IDE NetBeans para creación de código (getter y setter).

Por norma general, el nombre de las clases se define con la primera letra de su propio nombre en mayúscula. Si el nombre incluye más de una palabra, ésta se definirá también con la primera letra en mayúscula y sin separación con respecto a la anterior. El constructor deberá tener el mismo nombre que la clase.

```
private | public class Persona{
    //propiedades
    int i=0;
    char letra='a';
    //métodos
    public int sumamos(int x1, int x2){
        return x1+x2;
    }
}
```

}

La definición completa de un método de una clase en Java es:

[private|protected|public] [static] [abstract] [final] [native] [synchronized]  
 TipoDevuelto NombreMétodo ( [tipo1 nombre1[, tipo2 nombre2 ]... ] )  
 [throws excepción1 [,excepción2]... ]

#### 4.4.1 Sintaxis de constructores en Java

Los constructores de una clase en Java se diferencian en los parámetros con los que se llaman desde la instrucción *new*. No devuelven nada y se llaman como la clase.

```
private | public class Persona{
    //propiedades
    int i=0;
    char letra='a';

    //constructores
    //un primer constructor sin parámetros
    public Persona(){
        i=5;
        ...
    }
    //un segundo constructor con parámetros
    public Persona(int i){
        this.i = i;
    }

    //métodos
    public int sumamos(int x1, int x2){
        return x1+x2;
    }
}
```

## 4.5 Encapsulación, ocultamiento y visibilidad

En este apartado hablaremos de los tres conceptos básicos bajo los que se rigen las clases: la **encapsulación**, la **ocultación** y la **visibilidad**.

- **Encapsulación.** Hace referencia al modo en que se organiza la estructura de una clase. Es el principio que controla el acceso a sus atributos y métodos. Su objetivo es mantener el orden de los componentes de la clase y facilitar su uso.
- **Ocultamiento.** Se trata del concepto de ocultar las funciones y atributos de una clase que sólo se usan a nivel interno y que, por extensión, no tienen necesidad de ser accesibles externamente. Este proceso ayuda a que la clase sea más segura, limitando el acceso a las funciones para las que ha sido diseñada.
- **Visibilidad.** Consiste en definir los parámetros que, a su vez, definirán el ocultamiento de los elementos de una clase. Existen tres niveles de acceso a los atributos y métodos de una clase:
  - **Público.** Cualquier clase puede acceder al elemento.
  - **Protegido.** Sólo las clases hijas o derivadas pueden acceder al elemento.
  - **Privado.** El elemento está reservado únicamente para la clase que lo contiene.

### 4.5.1 Sintaxis de visibilidad en Java

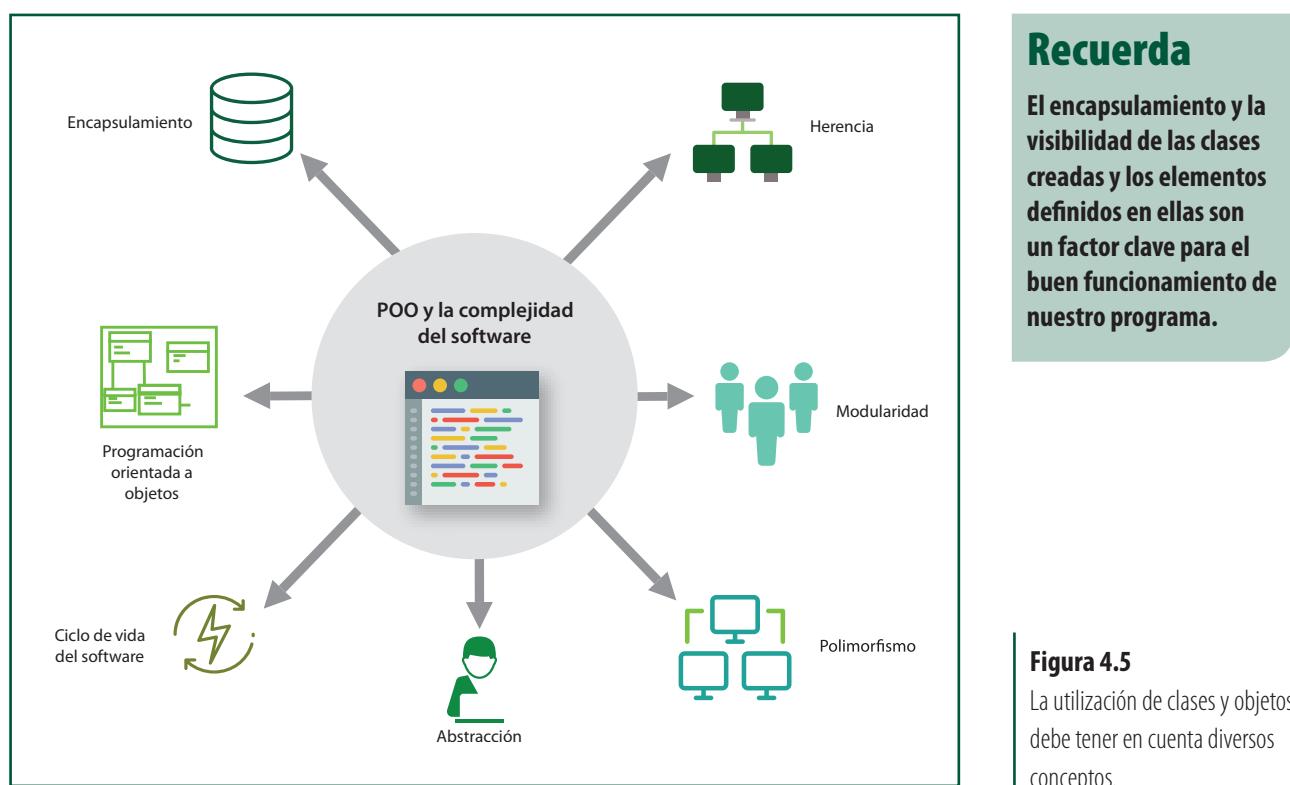
```
private | public class Persona{
    //propiedades
    public int i = 0; //pública
    private char letra = 'a'; //privada sólo accesible a la clase
    protected double d = 0.; //protegida sólo por clases heredadas
    y por otras en el paquete
    //constructores
    //un primer constructor sin parámetros
    public Persona(){
        i=5;
        ...
    }
    //un segundo constructor con parámetros
    public Persona(int i){
        this.i = i;
    }

    //métodos
    public int sumamos(int x1, int x2){
        return x1+x2;
    }
}
```

## 4.6 Utilización de clases y objetos

El **uso de clases y objetos** define el proceso a través del cual podremos agrupar, organizar e instanciar elementos en el código de nuestras aplicaciones (Figura 4.5). Lo primero que deberemos tener en cuenta a la hora de crear un objeto es si éste hará referencia a un concepto concreto o si podrá pertenecer a un grupo más amplio de elementos con características comunes. En este caso es cuando toman especial relevancia los conceptos de **encapsulación** y **empaqueado de clases**. Si nuestro objeto representa un elemento conciso o aislado, él mismo representará a su clase; de lo contrario, estaríamos hablando de un objeto derivado de un concepto más general que formaría parte de un grupo más amplio de elementos. En este caso se le atribuye al objeto la categoría de **subclase**.

En cuanto al método de uso en el código de clases y objetos es el mismo, la **instanciación**. Recordemos que la **instanciación** es el proceso a través del cual podremos generar un objeto en el código y utilizar sus funciones para que intervengan en la ejecución de nuestro programa. Una vez el objeto o la clase están instanciados en el código podremos proceder a aplicar sus métodos y propiedades para que intervengan en el curso del programa, bien sea de manera directa, o bien entre ellos a través de mensajes.



#### 4.6.1 Sintaxis de utilización de clases y objetos en java

```
class Persona{
    //propiedades
    public int i = 0; //pública
    private char letra = 'a'; //privada sólo accesible a la clase
    protected double d = 0.; //protegida sólo por clases
    heredadas y por otras en el paquete
    //constructores
    //un primer constructor sin parámetros
    public Persona(){
        i=5;
        ...
    }
    //un segundo constructor con parámetros
    public Persona(int i){
        this.i = i;
    }
    //métodos
    public int sumamos(int x1, int x2){
        return x1+x2;
    }
}
class Coche{
    int k = 0;
    Persona conductor = new Persona(); // creación del objeto
    condutor a partir de la clase Persona
    k = conductor.sumamos(3,4); // utilización del método
    sumamos del objeto conductor
    ...
}
```

#### 4.7 Utilización de clases heredadas

En **POO**, la **herencia** es el mecanismo fundamental que permite la reutilización y extensibilidad de un programa. A través de ella, los diseñadores pueden construir nuevas clases partiendo de una jerarquía de clases ya existente evitando el rediseño y la modificación de las mismas.

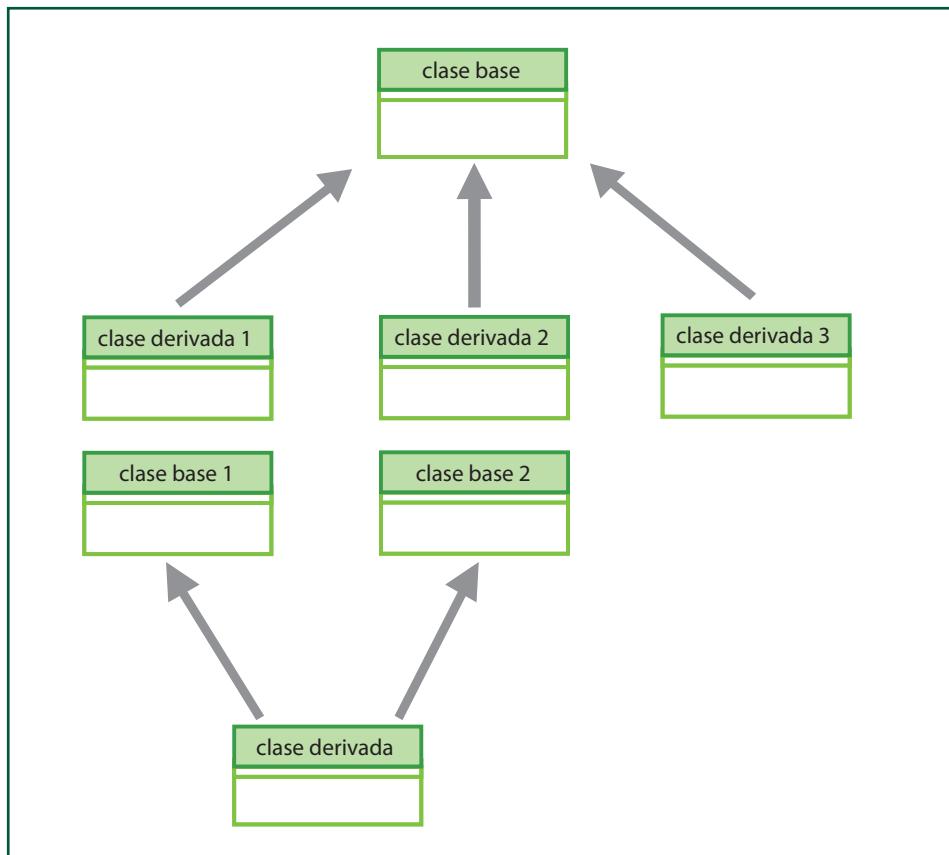
La **herencia** facilita la creación de objetos a partir de otros ya definidos a través de la obtención de características (métodos y atributos) comunes.

Con la **herencia** se establecen relaciones entre las clases generales y sus derivadas o más específicas. Por ejemplo, si declaramos una clase párrafo derivada de una clase texto, todos los métodos y variables asociadas con la clase texto, serán automáticamente heredados por la subclase párrafo.

La **herencia** es uno de los mecanismos más importantes que posee la **POO**, por medio del cual una clase puede derivarse de otra, de modo que se extienda su funcionalidad (Figura 4.6).

Según el lenguaje de programación, podemos encontrar dos tipos de herencias:

- **Simple.** Una clase puede derivar únicamente de otra clase.
- **Múltiple.** Una clase puede extender las características de una o más clases. Algunos lenguajes, como Java o C#, no permiten este tipo de herencia y otros como C++ sí.



**Figura 4.6**  
Ejemplo de herencia simple  
(primera) y compuesta  
(segunda).

### 4.7.1 Sintaxis de herencia de clases

```

class Persona{
    //propiedades
    public int i = 0; //pública
    private char letra = 'a'; //privada sólo accesible a la clase
    protected double d = 0.; //protegida sólo por clases
    heredadas y por otras en el paquete
    //constructores
    //un primer constructor sin parámetros
    public Persona(){
        i=5;
        ...
    }
    //un segundo constructor con parámetros
    public Persona(int i){
        this.i = i;
    }
    //métodos
    public int sumamos(int x1, int x2){
        return x1+x2;
    }
}
class Alumno extends Persona{
    public double nota = 0;
    public int sumamos(int x1, int x2){
        int a = super.sumamos(x1,x2);
        System.out.println("la suma es: " + a)
        return a;
    }
}

```

En este caso, *Alumno* hereda de *Persona*. Por lo tanto, tiene las mismas propiedades y los mismos métodos con las salvedades siguientes:

- Se ha añadido la propiedad *nota*.
- Se ha reescrito el método *sumamos*; que, además de hacer la suma (llamando a la clase de la que hereda con *super.sumamos x1, x2*), también imprime el resultado.

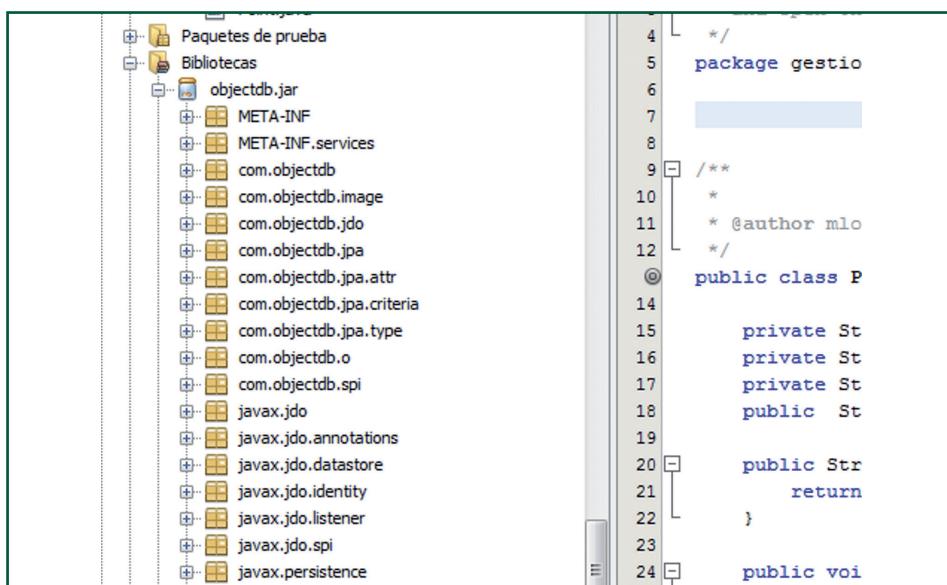
#### Recuerda

**Agrupando las clases en paquetes, el código resultante del programa podrá ser reutilizado en futuros proyectos y aplicaciones de funcionalidades similares.**

### 4.8 Empaquetado de clases

El **empaquetado de clases** (*class package*) es el proceso que permite agrupar las distintas partes de un programa cuya funcionalidad tienen elementos comunes (Figuras 4.7 y 4.8). Con el uso de paquetes de clases obtendremos las siguientes ventajas con respecto a la programación de nuestras aplicaciones:

- Agrupamiento de clases con características comunes creando una estructura organizativa del código en función de grupos distribuidos según sus funciones.
- Reutilización de código. Al agrupar un conjunto de clases en un paquete, éste podrá ser reutilizado en futuros proyectos, incluso por otros usuarios que quieran desarrollar aplicaciones de funcionalidades similares.
- Mayor seguridad gracias a los niveles de acceso. Se limita el acceso sólo a las acciones ejecutables, no a las que se usan a nivel interno de las clases. Esto evitará que ciertas funciones interfieran entre ellas o que se ejecuten acciones en puntos inadecuados del código fuente.



The screenshot shows a Java development environment. On the left, there's a tree view of packages and libraries. Under 'Bibliotecas', there's a 'objectdb.jar' entry with many sub-packages like 'META-INF', 'com.objectdb', etc. On the right, a code editor window displays a Java class named 'P'. The code includes annotations like '@author mlo' and methods like 'public String return()'.

```

4  /*
5   * package gestio
6
7
8
9  /**
10  *
11  * @author mlo
12  */
13 public class P
14
15     private St
16     private St
17     private St
18     public St
19
20     public Str
21         return
22     }
23
24 public voi

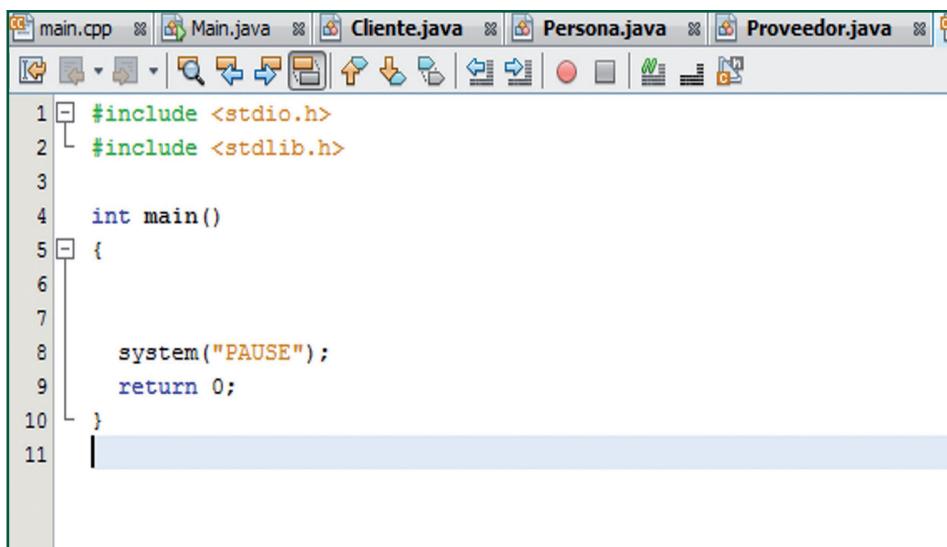
```

## Para saber más

**La herencia entre clases ayuda a definir nuevas clases (subclases) aprovechando las características de una clase más general. Al mismo tiempo, podremos crear también subclases a partir de clases derivadas, lo que nos permitirá alcanzar niveles más específicos para funcionalidades de nuestros objetos.**

**Figura 4.7**

Diferentes paquetes de ejemplo en Java.



The screenshot shows a C++ development environment. The code editor displays a main.cpp file. It contains standard library includes ('#include <stdio.h>', '#include <stdlib.h>'), a main function, and a system call to pause the program ('system("PAUSE");').

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6
7
8     system("PAUSE");
9     return 0;
10}
11

```

**Figura 4.8**

En C++, las librerías contienen funciones o clases a partir de *includes*.

## Resumen

Una clase es un tipo de datos que se compone de un conjunto de variables denominadas miembros o propiedades, y de un conjunto de funciones denominadas métodos.

Las propiedades y los métodos se pueden clasificar según su visibilidad en públicos y privados, es decir, accesibles desde fuera de la clase o únicamente desde dentro, respectivamente. De esta forma, se dice que el diseñador encapsula la clase, publicando aquellas propiedades (en ortodoxia ninguna) y métodos que interese, definiendo un comportamiento de la clase hacia el exterior.

Además de esta clasificación, las propiedades y los métodos pueden ser estáticos. Una propiedad estática es una variable definida a nivel de clase, es decir, compartida por todos sus objetos. Un método estático es aquel que puede ser llamado con el formato "nombre de clase.nombre de método" sin necesidad de instanciar ningún objeto.

Una clase puede heredar de otra (superclase) mediante la palabra reservada "extends". La clase heredará las propiedades y métodos de la superclase, excepto los métodos constructores. La clase heredera puede redefinir los métodos de la superclase y puede llamar a los métodos de la superclase con la palabra reservada super.

Cuando aplicamos herencia, toma relevancia el tipo de visibilidad protected, que permite que los miembros sean accesibles por las clases derivadas.

Se denomina empaquetado de clases en Java a la creación de un archivo (paquet) de clases que contiene un conjunto de clases con un objetivo común.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Las variables miembro y los métodos definidos en una clase serán compartidos o heredados por todos los objetos que de ella deriven.
2. Un principio de la POO es la ocultación, que indica que para modificar o acceder a una variable propia de una clase es preciso utilizar un paso intermedio llamado instanciación.
3. Cuando hablamos de atributos de una clase, nos referimos a todo dato destinado a ser compartido por todos los objetos que de ella deriven.
4. Una propiedad estática (*static*) es aquella que es única a nivel de objeto, no de clase.
5. La visibilidad consiste en definir los parámetros que, a su vez, definirán el ocultamiento de los elementos de una clase.
6. Los métodos privados (*private*) solo pueden ser llamados a nivel interno de la clase. Hacen referencia a funciones propias de un objeto, algo que este no tiene por qué compartir con otras clases u objetos derivados.
7. La herencia facilita la creación de objetos a partir de otros ya definidos a través de la obtención de características (métodos y atributos) comunes.

**Completa las siguientes afirmaciones:**

8. Los \_\_\_\_\_ sirven para conocer el valor de un atributo en concreto. Es un método de solo lectura, que se limita a devolver el \_\_\_\_\_ de un único dato en concreto y, en ningún caso, podrá modificarlo.
9. Los constructores de una clase en Java se diferencian en los \_\_\_\_\_ con los que se llaman desde la instrucción \_\_\_\_\_. No devuelven nada y se llaman como la \_\_\_\_\_.

10. Los tres conceptos básicos bajo los que se rigen las clases son los siguientes: la \_\_\_\_\_, la \_\_\_\_\_ y la \_\_\_\_\_.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 5. LECTURA Y ESCRITURA DE LA INFORMACIÓN

Cuando se empieza en el mundo de la programación, lo primero que se debe tener claro es que un programa es la resolución de un algoritmo cuyo esquema más general puede ser:

- Entrada de datos
- Transformación de datos
- Salida de datos

Por cuestiones pedagógicas, el aprendizaje de la programación pasa por un primer momento donde la entrada de datos es siempre por el teclado y la salida es en la consola (en la pantalla).

Pero existe un gran número de situaciones donde nos interesa cambiar la entrada y/o la salida de datos por ficheros. Es decir, **leer los datos** desde un fichero del sistema y **escribir los resultados** de nuestro programa en un fichero del sistema.

Pero esto no es todo, pues nos dejaríamos otro gran número de situaciones donde la entrada de datos debe realizarse desde un formulario en una ventana del sistema operativo y, en la salida de nuestro programa, deseamos mostrarla en esa u otra ventana del sistema operativo.

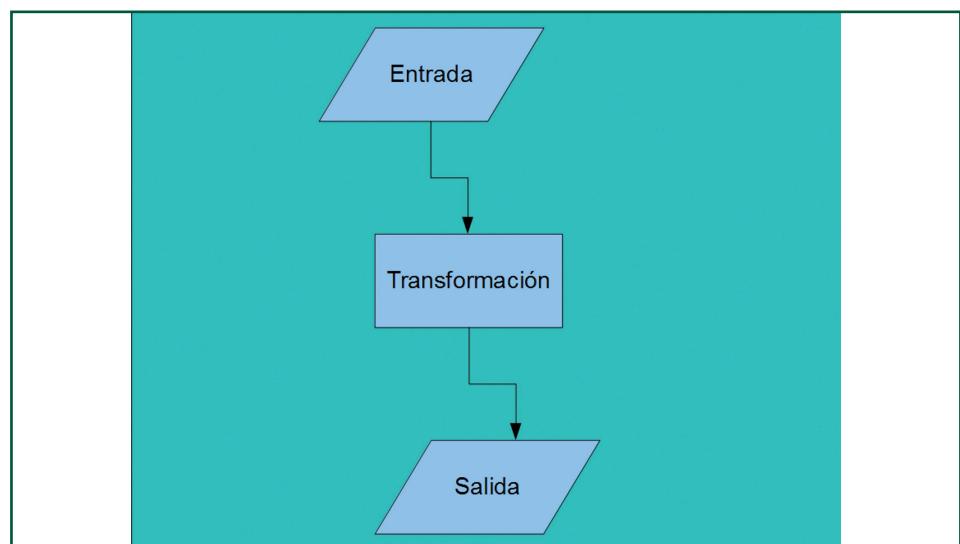
En esta unidad vamos a abordar estos dos tipos de entrada y salida de datos. En primer lugar, nos dedicaremos a estudiar la entrada y salida de datos sobre ficheros.

En Java, la comunicación entre nuestro programa y los ficheros se efectúa abriendo un flujo de datos (stream) y todo ello se gestiona con las clases `InputStream` y `OutputStream` (`Reader` y `Writer` para caracteres). Con estas clases y sobre todo, las clases que heredan de ellas.

En segundo lugar, trataremos la entrada y salida gráfica, es decir hacia y desde ventanas, estudiando además los formularios, sus controles y eventos, como por ejemplo, un botón y el evento “al hacer clic”. En Java lo haremos mediante las clases contenidas en paquetes de AWT (*Abstract Window Toolkit*), un sistema que permite a nuestros programas Java ser independiente del sistema de gráficos y ventanas del sistema operativo.

## 5.1 Concepto de flujo

En programación, definimos **flujo** (*stream*) a la conexión existente entre un programa y una fuente o destino de datos. Un **flujo** define el protocolo usado por los lenguajes de programación para controlar y tratar la entrada (*input*) y salida (*output*) de datos (Figura 5.1). Los **flujos de entrada** (*input*) controlan los datos que fluyen hacia un programa para ser procesados. Los **flujos de salida** (*output*) controlan los datos que fluyen desde un programa después de ser procesados. En programación, entendemos por **dato** aquella parte representativa de información proporcionada u obtenida del programa. Como ejemplo de entrada de datos podríamos tomar instrucciones del teclado o datos de un archivo, y como ejemplo de salida de datos podríamos tomar la información mostrada a través de la pantalla o un documento enviado a impresora. De este modo se unifica y simplifica la manera en la que el programador puede usar los dispositivos de entrada /salida.



**Figura 5.1**

Esquema conceptual de entrada y salida de datos en un programa informático.

## 5.2 Tipos de flujos: flujos de bytes y de caracteres

Existen dos tipos básicos de flujos de datos y ambos están destinados a permitir el uso de ficheros externos a nuestro programa:

- **Flujos de bytes.** Incluyen los métodos e instrucciones usados para manipular datos binarios, es decir, legibles sólo por un ordenador (por ejemplo un fichero *.exe*).
- **Flujos de caracteres.** Incluyen los métodos e instrucciones usados para manipular datos legibles por humanos (por ejemplo un fichero *.txt*).

## 5.3 Flujos predefinidos

En Java, los **flujos predefinidos**, o estándar, definen procesos a nivel interno que permanecen abiertos a lo largo de la ejecución de un programa. Dichos procesos están dispuestos tanto a recibir datos de entrada como a dirigir datos de salida hacia el dispositivo (pantalla, impresora...) o elemento (archivo) designado. Por defecto, el dispositivo de flujo estándar de entrada es el teclado, y la clase y objeto para referirnos a él es **System.in**. Asimismo, el flujo estándar de salida por defecto es la pantalla, y para referirse a él se usa la clase y objeto **System.out**. Un tercer objeto de la clase System, llamado **System.err**, se encarga de recoger y gestionar los posibles errores generados durante el proceso de flujos de entrada y salida de datos.

## 5.4 Clases relativas a flujos

Existen en Java una serie de clases dedicadas a controlar los flujos de entrada y salida de datos, y que se conocen con el nombre de **clases relativas a flujos**. Las clases diseñadas para tales propósitos son cuatro: dos para flujos de entrada y salida de bytes y dos para flujos de entrada y salida de caracteres. De ellas se derivan otras subclases destinadas a realizar funciones más específicas. Veamos cuáles son:

- **Clases relativas a flujos de bytes:**

- **InputStream.** Engloba las funciones y clases encargadas de leer los flujos de entrada de bytes y convertirlos en los denominados caracteres *Unicode* para que puedan ser interpretados por el ordenador.
- **OutputStream.** Engloba las funciones y clases encargadas de enviar los flujos de salida de bytes hacia los dispositivos designados, y convertirlos en caracteres comprensibles por el cerebro humano.

- **Clases relativas a flujos de caracteres:**

- **Reader.** Implementa todas las funcionalidades de lectura de flujos de caracteres para que estos puedan ser interpretados y procesados por el programa.
- **Writer.** Implementa todas las funcionalidades de escritura de flujos de caracteres para que estos puedan ser mostrados por pantalla o almacenados en archivos.

### Recuerda

**El flujo es el concepto que define el proceso de entrada y salida de información en un programa informático para unificar el acceso a toda la entrada/salida.**

## 5.5 Utilización de flujos

La **utilización de flujos** define un protocolo a seguir a la hora de trabajar con flujos de datos. Este protocolo varía según sea el flujo de lectura o de escritura:

- **Lectura de datos:**

- Creación del llamado objeto *stream*, que se encarga de abrir un flujo a una fuente entrada de datos (teclado, archivo, *socket*).
- Lectura de los datos desde el inicio de su flujo hasta su fin.
- Fin del proceso, uso del llamado método *close*, que se encarga de cerrar el flujo de datos y terminar el proceso, liberando así los sectores de memoria destinados a él.

- **Escritura de datos:**

- Creación de un objeto *stream* para abrir un flujo a una fuente salida de datos (pantalla, archivo).
- Escritura de los datos desde el inicio hasta el final de su flujo.
- Fin del proceso con el llamado método *close*, que se encarga de cerrar el flujo de datos y terminar el proceso liberando así los sectores de memoria utilizados para él.

Los flujos estándar son gestionados a nivel interno por el sistema: él se encarga de su apertura y de su cierre según sea necesario.

## 5.6 Entrada desde teclado

En términos generales, se considera el **teclado** como el dispositivo principal de entrada de datos al sistema o al programa.

Los datos enviados desde el teclado del ordenador se gestionan con el objeto *System.in* de la clase *InputStream*. Este objeto será el encargado de tratar los flujos de *bytes* recogidos cada vez que se pulse una tecla, y de traducirlos a caracteres interpretables por el ordenador para que puedan ser procesados.

En lenguaje Java se incluyen ya una serie de clases designadas para tales efectos, sin embargo tenemos también la posibilidad de crear nuestras propias clases de control a partir de ellas, para obtener resultados más específicos cuando se detecte una entrada de teclado en nuestro programa. La combinación de todo esto hace que las instrucciones para leer desde el teclado sean como en el ejemplo siguiente:

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

## 5.7 Salida a pantalla

Generalmente, se considera la **pantalla** como el dispositivo principal de salida de datos del sistema o programa.

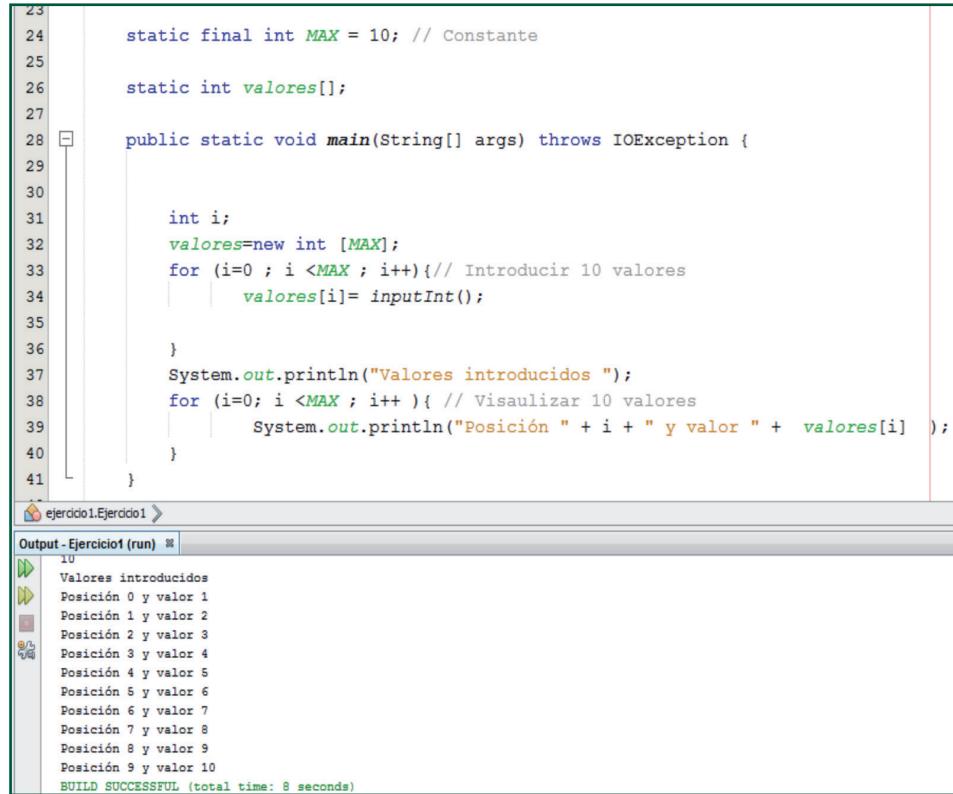
Los datos enviados hacia la pantalla por el programa se gestionan mediante el objeto `System.out` perteneciente a la clase `OutputStream`. Este objeto será el encargado de tratar los flujos de bytes enviados, y de traducirlos a caracteres o resultados interpretables por una persona.

Igual que para los flujos de entrada, en lenguaje Java se incluyen una serie de clases designadas para tales efectos; sin embargo, tenemos también la posibilidad de crear nuestras propias clases de control a partir de ellas.

Por defecto, los flujos de datos de salida se muestran por pantalla a través de la **consola** (Figura 5.2) de nuestro sistema operativo. No obstante, una vez terminado el programa, podremos definir que dichos datos sean mostrados por pantalla representados de un modo más visual (como, por ejemplo, en gráficos) e incluso ofrecer la posibilidad de que sean manipulados por el usuario a través de entradas de teclado.

En la práctica, lo más habitual en Java para escribir en la consola es tal y como se muestra en el siguiente ejemplo:

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



The screenshot shows an IDE interface with two main sections. The top section is a code editor for a file named 'ejercicio1.Ejercicio1.java'. The code is as follows:

```

23
24     static final int MAX = 10; // Constante
25
26     static int valores[];
27
28     public static void main(String[] args) throws IOException {
29
30         int i;
31         valores=new int [MAX];
32         for (i=0 ; i <MAX ; i++){// Introducir 10 valores
33             valores[i]= inputInt();
34
35         }
36         System.out.println("Valores introducidos ");
37         for (i=0; i <MAX ; i++){ // Visualizar 10 valores
38             System.out.println("Posición " + i + " y valor " + valores[i] );
39         }
40     }
41 }
```

The bottom section is a terminal window titled 'Output - Ejercicio1 (run)' showing the program's output:

```

ejercicio1.Ejercicio1 >
Output - Ejercicio1 (run) ✘
10
Valores introducidos
Posición 0 y valor 1
Posición 1 y valor 2
Posición 2 y valor 3
Posición 3 y valor 4
Posición 4 y valor 5
Posición 5 y valor 6
Posición 6 y valor 7
Posición 7 y valor 8
Posición 8 y valor 9
Posición 9 y valor 10
BUILD SUCCESSFUL (total time: 8 seconds)
```

## Para saber más

Combinando las funciones que ofrecen la clases relativas a flujos, seremos capaces de crear nuevas funcionalidades adaptadas a las necesidades de nuestros programas.

**Figura 5.2**

Ejemplo de uso de la consola para verificar una aplicación.

# Recuerda

**Se considera que el teclado y la pantalla son los dispositivos principales de entrada y salida de datos en un programa informático; sin embargo, en este proceso pueden intervenir muchos más dispositivos, como la impresora, una webcam o el ratón.**

## 5.8 Aplicaciones del almacenamiento de información en ficheros

La filosofía de **almacenar información en ficheros** externos a nuestro programa es, a parte de la lógica necesidad de poder guardar unos datos de manera que no sean volátiles y que sean accesibles para futuros usos, la de separar la programación de la información (Figura 5.3). De este modo, pueden modificarse los datos de entrada a nuestro programa independientemente de cuál sea su código. Imaginemos que tenemos un programa diseñado para mostrar una galería de fotos. Para indicar al programa cuál es la ubicación de cada uno de los archivos de imagen, deberemos especificar una ruta para cada uno de ellos.

Una primera posibilidad es la de incluir la ruta de cada imagen dentro del mismo código. De este modo, el programa no dejará de ser funcional, pero su usabilidad y mantenimiento perderán eficacia, ya que cualquier modificación de la información, como el cambio del nombre de un archivo, afectará directamente al código de la aplicación. De otro modo, si almacenamos la lista de rutas en un archivo de texto independiente, realizar el mantenimiento de la aplicación se limitará a modificar la información de dicho archivo; luego el código se encargará de procesarla según proceda.

```
<fe:Facturae xmlns:ds="http://www.w3.org/2000/09/xmldsig#" xmlns:fe="http://www.facturae.es/Facturae/2009/v3.2/Facturae">
  <FileHeader>
    <SchemaVersion>3.2</SchemaVersion>
    <Modality>I</Modality>
    <InvoiceIssuerType>EM</InvoiceIssuerType>
  <Batch>
    <BatchIdentifier>00000000000018</BatchIdentifier>
    <InvoicesCount>1</InvoicesCount>
    <TotalInvoicesAmount>
      <TotalAmount>63.13</TotalAmount>
    </TotalInvoicesAmount>
    <TotalOutstandingAmount>
      <TotalAmount>63.13</TotalAmount>
    </TotalOutstandingAmount>
    <TotalExecutableAmount>
      <TotalAmount>63.13</TotalAmount>
    </TotalExecutableAmount>
    <InvoiceCurrencyCode>EUR</InvoiceCurrencyCode>
  </Batch>
</FileHeader>
<Parties>
  <SellerParty>
    <TaxIdentification>
      <PersonTypeCode></PersonTypeCode>
      <ResidenceTypeCode>R</ResidenceTypeCode>
      <TaxIdentificationNumber>A82735122</TaxIdentificationNumber>
    </TaxIdentification>
    <LegalEntity>
      <CorporateName>Company Comp SA</CorporateName>
      <TradeName>Comp</TradeName>
    </LegalEntity>
  </SellerParty>
</Parties>
```

### Figura 5.3

## Ejemplo de un archivo XML.

## 5.9 Ficheros de datos. Registros

### 5.9.1 Registros del sistema

Las versiones de Java a partir de la versión 1.4 añaden una funcionalidad que permite controlar, dar formato y publicar mensajes a través de los llamados re-

gistros (*log*). Los registros pertenecen a un paquete de clases llamado ***java.util.logging***. Los registros pueden ser usados para lanzar mensajes de información, estados de los datos o incluso errores ocurridos durante la ejecución del programa. Estos procesos pueden beneficiar a los usuarios de la aplicación sea cual sea su perfil. El proceso de utilización de los registros de Java es muy simple, sólo hay que seguir los siguientes pasos (Figura 5.4):

- Incluir las sentencias de importación para el paquete de clases *logging*.
- Creación de una referencia a la clase *Logger*.
- Indicar el mensaje que desea mostrarse y su tipología.

```

1  import java.io.*;
2  import java.util.logging.*;
3
4  public class QuintLog {
5      public static void main(String[] args) {
6          try {
7              FileHandler hand = new FileHandler("vk.log");
8              Logger log = Logger.getLogger("log_file");
9              log.addHandler(hand);
10             log.warning("Doing carefully");
11             log.info("Doing something...");
12             log.server("Doing strictly");
13             System.out.println(log.getName());
14         }
15         catch(IOException e) {}
16     }
17 }
18

```

**Figura 5.4**

Ejemplo de implementación de la clase *Logger* en código Java.

### 5.9.2 Registros de datos propios

En muchas aplicaciones es necesario almacenar la información en un formato de texto determinado para que sea legible y se pueda recuperar a posteriori. Utilizando las clases de gestión de flujos es relativamente fácil; sin embargo, es necesario definir previamente un formato que permita su posterior lectura. Existen gran variedad de formas, pero las más usuales son las siguientes:

- Usar un tamaño fijo de registro, de modo que todos los registros ocupen lo mismo.
- Usar un tamaño variable, y separadores de campo y de registro.

Suponiendo que se desee guardar la información de la clase Pieza, con los campos, código, nombre y precio, las dos opciones serían:

### Tamaño fijo

Ejemplo de registros de tamaño fijo

000001ConectorRJ45	0001.45
000002CableUTP6e	0009.27
009871ImpresoraRh99	0123.67

### Recuerda

**El almacenado de información en archivos externos de nuestro programa nos proporcionará un mayor control de nuestras aplicaciones al dejar de depender únicamente del código fuente.**

### Tamaño variable y separadores

1#ConectorRJ45#.	45
2#CableUTP6e#	9.27
9871#ImpresoraRh99#	123.67

En este caso se usa el símbolo # como separador de campos y el salto de línea como separador de registros.

## 5.10 Apertura y cierre de ficheros. Modos de acceso

Igual que en los procesos designados para controlar la apertura, proceso previo necesario para cualquier acceso a un fichero, y cierre de flujos de información, existen también unos protocolos que rigen la apertura y cierre de los ficheros externos usados en nuestro programa.

A través del paquete de clases **FileReader**, Java proporciona toda una serie de funciones destinadas a la apertura, lectura y cierre de ficheros. Por medio de una ruta definida por un archivo alojado en nuestro disco duro, o por una URL de Internet, se indica a la clase **FileReader** cuál es el origen de los datos. Otras clases y funciones como **BufferedReader** o **readLine** se encargarán de recoger en forma de datos la información contenida en los ficheros y almacenarla en variables de nuestro programa.

Una vez terminado el proceso, finalizaremos cerrando el fichero y liberando de la memoria todos los recursos que habían sido utilizados para su tratamiento.

## 5.11 Escritura y lectura de información en ficheros

Gracias a las clases **FileReader**, **FileWriter** y algunas derivadas de **InputStream** y **OutputStream** de Java, seremos capaces de acceder al contenido almacenado en archivos externos a nuestro programa para obtener o modificar su información (Figura 5.5).

```

package ejem17;
import java.io.*;

public class copyFile {

    public static void main(String[] args) {
        File origen=new File(args[0]);
        File destino=new File(args[1]);
        FileInputStream in=null;
        FileOutputStream out=null;
        try {
            in=new FileInputStream(origen);
            out=new FileOutputStream(destino);
            byte[] buffer=new byte[4096];
            while (true) {
                int n=in.read(buffer);
                if (n>0) {
                    out.write(buffer,0,n);
                } else {
                    System.out.println(args[0]+" copiado a "+args[1]);
                    break;
                }
            }
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            try {
                if (in!=null) {
                    in.close();
                }
                if (out!=null) {
                    out.close();
                }
            } catch (IOException e1) {
                System.out.println(e1);
            }
        }
    }
}

```

**Figura 5.5**

Ejemplo de código Java con implementación de lectura y escritura de un archivo.

Este proceso nos permitirá tener información almacenada externamente para poder acceder a ella en puntos determinados del programa. Tal y como explicábamos en apartados anteriores, este proceso nos permitirá separar la programación de la información, pudiendo modificar así los datos de entrada independiente- mente de cuál sea nuestro código. La tipología de archivos que podrá ser utilizada y manipulada por nuestros programas es muy diversa, puede tratarse de simples documentos de texto con extensión .txt, o archivos de información estructurada como un .xml, archivos de imagen, o incluso archivos ejecutables.

Cuando manejamos archivos de texto, se utilizan FileReader y FileWriter, junto con otras clases tipo Buffered para leer y escribir datos (en este caso texto). En el caso de otro tipo de ficheros, donde manejamos bytes, se utilizan las clases InputStream y OutputStream, además de sus derivadas.

## 5.12 Almacenamiento de objetos en ficheros. Persistencia. Serialización

La **serialización** es el proceso a través del cual pueden almacenarse objetos directamente en una secuencia de bytes. Se trata de un proceso basado en el

concepto de la **persistencia**, que permite la posterior reconstrucción de los objetos guardados como secuencias de *bytes* (Figura 5.6).

Las clases encargadas de controlar estos procesos son ***ObjectOutputStream*** y ***ObjectInputStream***. La primera se encarga de serializar el objeto convirtiéndolo en una secuencia de *bytes*, la segunda es la encargada de leer e interpretar la secuencia de *bytes* de un objeto y reconstruirlo.

Mediante estos métodos podremos realizar operaciones complejas con nuestros objetos como, por ejemplo, enviarlos a través de la red en formato de *bytes* y reconstruirlos de nuevo una vez lleguen a su destinatario.

**Figura 5.6**  
Código Java con ejemplo del proceso de serialización y persistencia.

```
/*
 * Escribe a archivo el objeto Empleado
 *
 * @param archivo Nombre de archivo
 * @throws IOException Excepción de entrada/salida
 */
public void serializar (String archivo) throws IOException {
    ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(archivo));
    salida.writeObject(this);
}
```

## 5.13 Utilización de los sistemas de ficheros

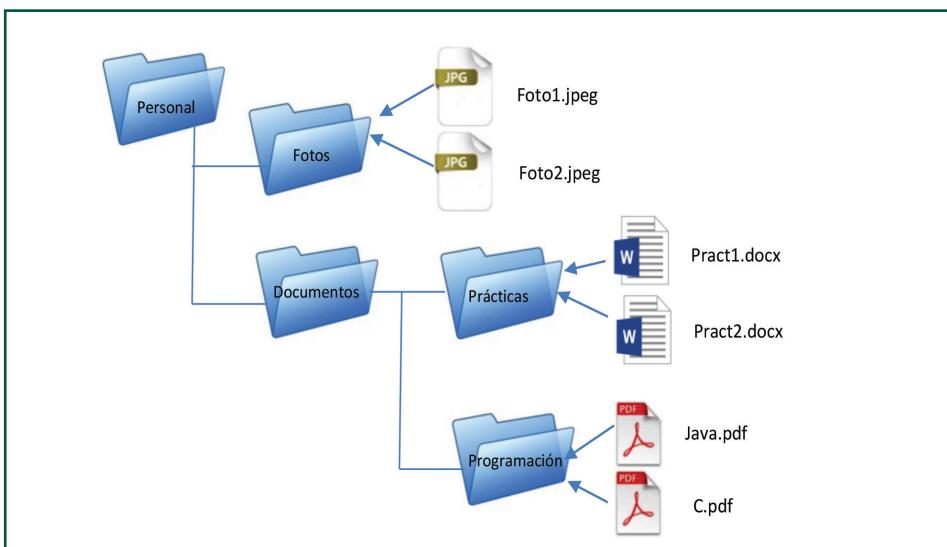
Un **sistema de ficheros** es la representación de todos los elementos incluidos en nuestro ordenador, ya sean unidades (en algunos sistemas operativos), carpetas, archivos, programas o dispositivos externos, como impresoras o escáneres (Figura 5.7).

Java incluye una serie de funcionalidades para poder interactuar con todos ellos. Igual que podemos detectar una entrada de teclado y registrar una acción a partir de ello, podremos también acceder a métodos que activen funciones de impresión, modificación de archivos o ejecución de otros programas complementarios de apoyo.

El sistema de archivos proporciona también al lenguaje la organización de nuestros archivos y directorios en el disco duro, de este modo tendremos la posibilidad de poder seleccionar un archivo concreto para que sea procesado en tiempo de ejecución del programa. Así se podrá escoger un archivo de una ruta, saber si es un archivo o una carpeta, el tipo de archivo, etc.

### Para saber más

**Los ficheros de registros son una herramienta muy valiosa en cuanto a diseño de aplicaciones se refiere. Suponen el complemento perfecto, ya sea para otros programadores que intervengan en su desarrollo como para los usuarios finales.**



**Figura 5.7**  
Ejemplo de sistema de archivos.

## 5.14 Creación y eliminación de ficheros y directorios

El lenguaje Java ofrece una serie de operativas que permiten la **creación y eliminación de ficheros** físicos en el disco duro. La clase encargada de llevar a cabo estos procesos se llama *File* y está incluida dentro del paquete de clases llamado **Java IO**. El método que deberemos utilizar es *createNewFile*, incluido dentro de las funcionalidades de dicha clase. El proceso es bien simple, tan sólo deberemos incluir dos parámetros: el primero indicará la ruta física donde se quiera crear el archivo; el segundo indicará el nombre que recibirá el archivo creado (Figura 5.8).

El único factor a tener en cuenta es que la ruta definida para la creación del archivo deberá existir previamente en nuestro sistema. Hay que señalar que la doble barra usada en la sentencia es la que le indicará a nuestro código la jerarquía o separación de los directorios en la ruta indicada. Una vez creado el fichero, bastará con invocar el método *createNewFile*, definiendo una estructura de excepción (*IOException*), que se encargará de recoger los posibles errores que aparezcan a lo largo del proceso.

```

1  ...
2  import java.io.File;
3  ...
4  File f = new File("un_path/un_fichero.txt");
5
6  if(f.canRead())
7      // El fichero existe y se puede leer.
8
9  if(f.canWrite())
10     // El fichero existe y se puede escribir en él.
11
12 if(f.canExecute())
13     // El fichero existe y se puede ejecutar.
  
```

**Figura 5.8**  
Código Java que además de crear identifica los permisos.

Como se ha explicado, en Java, la gestión de un fichero o de un directorio se efectúa mediante la clase *File*. Además del método explicado, otra manera de trabajar es con los constructores que podemos ver en los ejemplos siguientes:

```
File f1 = new File("c:\\windows\\notepad.exe"); // La barra \' se escribe '\\'
File f2 = new File("c:\\windows"); // Un directorio
```

La propiedad estática *File.separator* nos proporciona el carácter separador del sistema de archivos donde se está ejecutando nuestro programa. En el ejemplo anterior, si queremos que nuestro programa funcione en Linux y en Windows, deberíamos utilizar esta propiedad en cualquier referencia a una ruta del sistema de ficheros. Por ejemplo:

```
File f1 = new File("directorio"+File.separator+"directorio"+ ...
```

Otra utilidad que nos ofrece Java, es la ventana de diálogo del sistema de ficheros para seleccionar un fichero o directorio concretos. La gestión de estas ventanas de diálogo en Java se efectúa mediante la clase *FileDialog* del paquete *java.awt*.

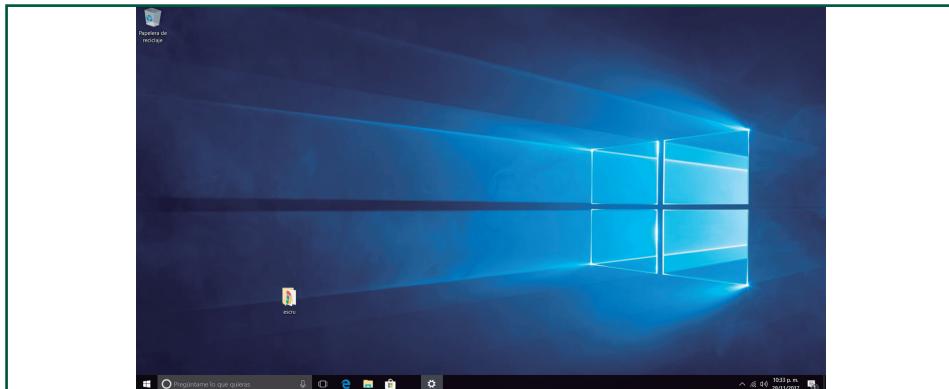
Una vez se ha creado el objeto *File* con éxito, ya sea un directorio o un fichero, el método "delete" permite la eliminación de dicho directorio o fichero del disco. El siguiente ejemplo elimina el fichero "a.txt" del disco, controlando si se ha efectuado con éxito. En el caso del directorio, se procedería igual:

```
File fic = new File("a.txt");
if (fic.delete())
    System.out.println("El fichero ha sido borrado satisfactoriamente");
else
    System.out.println("El fichero no puede ser borrado por algún motivo");
```

## 5.15 Creación de interfaces gráficos de usuario utilizando asistentes y herramientas del entorno integrado

En un contexto de interacción persona-ordenador, la **interfaz gráfica de usuario** (IGU), se refiere al conjunto de herramientas tecnológicas que posibilitan, de una forma visual, la interacción con un sistema informático.

La **interfaz gráfica de usuario** (en inglés *Graphical User Interface*, GUI), a través de un conjunto de imágenes y objetos gráficos (iconos y ventanas), es capaz de representar la información y acciones disponibles en una aplicación informática.

**Figura 5.9**

Escritorio del sistema operativo Windows como ejemplo de interfaz gráfica de usuario.

La filosofía de una *GUI* es facilitar la interacción del usuario con el ordenador a través de acciones realizadas mediante la manipulación directa de los elementos en pantalla. Una *GUI* surge de la evolución de la consola de los primeros sistemas operativos, que representa la pieza fundamental de todo entorno gráfico. Como ejemplo de *GUI* podemos citar el escritorio del sistema operativo Windows (Figura 5.9). En general, una *GUI* es el entorno visual que se muestra cuando ejecutamos un programa informático.

En Java hay diversas utilidades, paquetes de clases o herramientas, que nos permiten implementar aplicaciones con entorno gráfico. Estas utilidades nos ofrecen la gran ventaja con respecto a otros lenguajes de poder hacer nuestro programa independiente del GUI del sistema operativo.

Entre otras, podemos utilizar la herramienta AWT (*Abstract Window Toolkit*) y Swing, aunque por cuestiones pedagógicas, para aprender a programar es conveniente utilizar AWT.

## 5.16 Interfaces

En programación, definimos **interfaz** a aquel entorno que permite establecer una conexión, a distintos niveles, entre ordenadores y dispositivos de cualquier tipo (Figura 5.10). El concepto de interfaz puede ser utilizado en diferentes contextos:

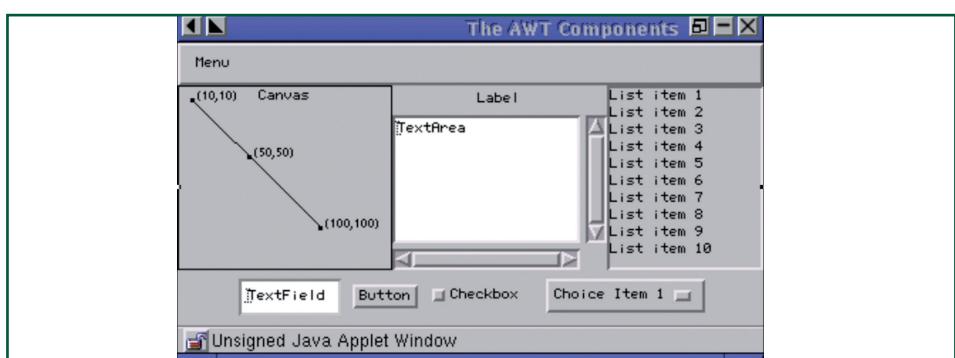
- **Interfaz como instrumento:** desde esta perspectiva, se considera una **interfaz** una extensión de nuestro cuerpo. Si tomamos el ejemplo de un ratón, podría definirse como un instrumento que extiende las funciones de nuestra mano y las lleva a la pantalla bajo la forma del cursor. Asimismo, la pantalla de un ordenador funciona como la interfaz que conecta el usuario con la información del disco duro.

## Recuerda

**Las Interfaces Gráficas de Usuario (GUI) son el soporte principal de comunicación sobre el que se apoyarán los usuarios para manejar nuestras aplicaciones. Su diseño debe ser intuitivo y estable.**

- **Interfaz como superficie:** en ocasiones se considera que una **interfaz** funciona como transmisora de información. La superficie de un objeto, por ejemplo, nos puede proporcionar información acerca de su forma, textura, color, etc.
- **Interfaz como espacio:** desde esta perspectiva, una **interfaz** es el entorno en el que tiene lugar la interacción con un programa, el espacio en el que se desarrollan las instrucciones y los intercambios de información.

En AWT de Java, el primer concepto para la interfaz gráfica es la separación entre contenedor y componente. La ventana es un contenedor (*Frame*) y un botón es un componente.



**Figura 5.10**

Imagen de una ventana de AWT con una muestra de algunos componentes de este entorno.

Los contenedores de AWT:

- Frame (classe `java.awt.Frame`)
- Panel (classe `java.awt.Panel`)

Como hemos visto, *Frame* es una ventana y *Panel* es un contenedor que podemos colocar dentro de una ventana con el simple propósito de agrupar componentes dentro de él.

Para organizar los contenidos (contenedores y componentes) dentro de una ventana se utilizan los *layout managers*. Pensemos que, a diferencia de otros entornos, no podemos ubicar libremente un componente dentro de una ventana, pero por el contrario, nuestro programa gráfico funcionará en cualquier sistema operativo.

Los principales *layout managers* de AWT:

- `java.awt.BorderLayout`
- `java.awt.FlowLayout`
- `java.awt.GridLayout`

Un *BorderLayout* divide el contenedor en cinco áreas, norte, este, oeste, sur y centro. En un *FlowLayout*, los componentes se van añadiendo a la derecha del anterior y, si no cabe en la anchura de la ventana, entonces se mostrará en la línea siguiente.

Un *GridLayout* corresponde a una tabla con n filas por m columnas. Cada componente que se añade se mostrará en la siguiente columna, empezando nueva fila cuando aquellas se hayan acabado.

Los componentes de AWT, básicamente controles, son:

- Etiqueta (classe `java.awt.Label`)
- Botón (classe `java.awt.Button`)
- Radiobutton / Checkbox (classe `java.awt.Checkbox`), agrupados o no en
  - un checkboxgroup (classe `java.awt.CheckboxGroup`)
- Lista (classe `java.awt.List`)
- Desplegable para seleccionar (classe `java.awt.Choice`)
- Campos de texto (classe `java.awt.TextField`)
- Áreas de texto (classe `java.awt.TextArea`)

## 5.17 Concepto de evento

El **concepto de evento** define toda aquella acción iniciada por un usuario durante la ejecución de un programa. Cuando el programa detecta un **evento**, se crea un **objeto** designado a realizar una acción relacionada con él. En Java, la clase dedicada al control de los eventos está incluida dentro del paquete `java.awt.Event` (Figura 5.11).

Existen muchos tipos de eventos para controlar los diferentes tipos de interacciones entre el usuario y el programa en ejecución. Veamos los principales:

- **Eventos de teclado (KeyEvent)**. Encargados de registrar acciones lanzadas a partir del pulsado de una tecla.
- **Eventos de texto (TextEvent)**. Encargados de registrar acciones lanzadas a partir del cambio de un texto en alguno de los campos de la aplicación.
- **Eventos de ratón (MouseEvent)**. Encargados de registrar acciones lanzadas a partir de acciones del ratón (clic, *rollOver*, *rollOut*...).
- **Eventos de foco (FocusEvent)**. Encargados de registrar acciones lanzadas a partir del cambio de la posición del foco del cursor en pantalla.
- **Eventos de acción (ActionEvent)**. Encargados de registrar acciones lanzadas a partir de la activación de elementos del programa, como, por ejemplo, pulsar un botón.

**Figura 5.11**

Esquema del proceso de escucha y activación de función de un controlador de eventos.

## 5.18 Creación de controladores de eventos

Un **controlador de eventos** es un método o función designado a lanzar una acción determinada cuando se registra un evento a lo largo de la ejecución de un programa. Los **controladores de eventos** se aplican directamente sobre los **objetos** definidos en nuestro código. Al aplicar un **controlador** sobre un **objeto**, éste recibe una función de escucha para dicho evento determinado. Por este motivo los controladores de eventos reciben el nombre de *listener*. Por ejemplo, si queremos detectar un clic de ratón en nuestra aplicación, deberemos recurrir a los eventos de ratón (*MouseEvent*) definidos en las clases del lenguaje. El proceso es el siguiente:

- Definición de un **objeto**.
- Asignación de un **controlador de evento** (*listener*) sobre el **objeto**.
- Definición de la función designada a realizar por el **evento**.

## 5.19 Generación de programas en entorno gráfico

La función de los **entornos gráficos de usuario** o GUI es ofrecer un abanico de herramientas para los programadores destinadas a diseñar aplicaciones de un modo visual e intuitivo.

La generación de programas a partir de entornos gráficos nos permite programar una aplicación al tiempo que evaluamos visualmente como será la experiencia final del usuario al que irá destinada.

Gracias al entorno visual que facilitan las GUI podremos crear fácilmente una aplicación con sólo arrastrar botones y otros componentes de las barras de opciones del entorno. No obstante, muchas veces el programador lo hace sin recurrir a estos entornos visuales, pues copia código que ya tiene realizado.

El esquema general para realizar una aplicación en Java AWT es:

- Crear un proyecto Java normal, como se ha hecho hasta ahora, por ejemplo con NetBeans. No crearemos la clase principal *Main.java*.
- Añadir una nueva clase, por ejemplo "Ventana" pero tipo Otros / de interfaz gráfico o AWT GUI y seleccionamos *Formulario Frame*. Hagamos que se cree también el paquete con el nombre que deseemos.
- Ya podemos ejecutar el programa. Si nos pide la clase principal, le decimos el nombre que le hemos dado a nuestro Formulario Frame. NetBeans nos ha añadido el código necesario.
- En este momento, podríamos añadir los componentes que nos hicieran falta, y los eventos sobre ellos. Podríamos cambiar el layout, añadir otros contenedores como paneles e incluso otras ventanas. En la parte práctica, haremos algún ejemplo de ello.

El código generado por NetBeans es el siguiente:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * Ventana.java
 *
 * Created xxx
 */

package as;

/**
 *
 * @author yyy
 */
public class Ventana extends java.awt.Frame {

    /** Creates new form Ventana */
    public Ventana() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
    */
}
```

## Para saber más

**El dominio de las clases de los controladores de eventos es uno de los pilares de la programación de aplicaciones informáticas, ya que nos permitirá tener un control preciso de todas las acciones que tengan lugar en un programa.**

```

 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent evt) {
            exitForm(evt);
        }
    });

    pack();
}// </editor-fold>

/** Exit the Application */
private void exitForm(java.awt.event.WindowEvent evt) {
    System.exit(0);
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Ventana().setVisible(true);
        }
    });
}

// Variables declaration - do not modify
// End of variables declaration

}

```

Podemos observar lo siguiente:

- Nuestra clase principal *Ventana* hereda de *Frame*.
- Se ha creado un método *main*, por donde entrará a ejecutarse nuestra aplicación. El código generado (*Runnable ...*) es para que la ventana se ejecute en un proceso diferente (*Thread*).
- El constructor se ejecuta cuando creamos el objeto anterior y, por lo tanto, se llama a *initComponents()*.
- El método *initComponents* añade un listener (el controlador de eventos, el “escuchador”) al contenedor *Ventana* y caza el evento *windowClosing* que se produce cuando cerramos la ventana. Al cazar el evento, llamamos a *exitForm*.

- El método *exitForm* llama *System.exit(0)*; con lo que el programa finaliza.

Pues esto que nos ha creado NetBeans contiene todos los aspectos a considerar en una aplicación gráfica. Se crea un objeto de tipo contenedor, se visualiza, se le añade un listener y se cazan eventos y además se ejecutan métodos cuando se ha cazado el evento. Y esto se puede ir repitiendo con otros contenedores y componentes. La estructura es la misma.

## Resumen

El **concepto de flujo** es una referencia amplia del proceso por el que pasa toda la información que interviene en nuestro programa, desde un punto de entrada hasta un punto de salida. Este proceso gestiona el estado de dicha información en cada uno de los puntos de nuestro programa. En todos los lenguajes de programación existe un control de dicho proceso; sin embargo, sólo en algunos de ellos, como Java, se incluye una serie de clases destinadas a la gestión de los datos por parte del programador.

Operar con ficheros no es una tarea sencilla de controlar; sin embargo, es de gran utilidad en cuanto a tratamiento de la información se refiere. El uso de archivos de apoyo en nuestros programas, tanto para la entrada como para la salida de datos, es una herramienta indispensable que debe dominar todo programador, sea cual sea el lenguaje en el que se especialice. El uso de ficheros nos ofrece unas ventajas inestimables tanto para organizar la información entrante en nuestro programa como para gestionar la salida de datos o el reporte de errores con respecto a los usuarios finales de la aplicación.

Gracias a los controladores de eventos podremos controlar de forma precisa qué acciones se desencadenan en puntos concretos de nuestro código, obteniendo así un acceso directo a posibles errores que puedan surgir en él.

Las interfaces gráficas de usuario (GUI) representan una herramienta muy versátil a la hora de diseñar aplicaciones, ya que, gracias al entorno visual que ofrecen, permiten programar de un modo sólido y, al mismo tiempo, evaluar la experiencia final y la usabilidad de la aplicación diseñada.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Los flujos de bytes incluyen los parámetros y las instrucciones usados para manipular datos binarios y datos legibles, es decir, legibles sólo por un ordenador.
2. Un flujo define el protocolo usado por los lenguajes de programación para controlar y tratar la entrada (*input*) y salida (*output*) de datos.
3. En Java, los flujos predefinidos, o estándar, definen procesos a nivel externo que permanecen abiertos a lo largo de la ejecución de un programa.
4. Las clases relativas a flujos son cuatro: dos para flujos de entrada y salida de *bytes* y dos para flujos de entrada y salida de caracteres.
5. OutputStream engloba las funciones y clases encargadas de enviar los flujos de salida de bytes hacia los dispositivos designados y convertirlos en caracteres comprensibles por el cerebro humano.
6. La serialización es el proceso a través del cual pueden almacenarse objetos directamente en una secuencia de código.
7. Los controladores de eventos se aplican de manera indirecta sobre los objetos definidos en nuestro código.

**Completa las siguientes afirmaciones:**

8. Existen dos tipos básicos de flujos de datos y ambos están destinados a permitir el uso de \_\_\_\_\_ externos a nuestro programa: flujos de \_\_\_\_\_ y flujos de \_\_\_\_\_.
9. A través del paquete de clases \_\_\_\_\_, Java proporciona toda una serie de funciones destinadas a la \_\_\_\_\_, \_\_\_\_\_ y cierre de ficheros.

10. El concepto de evento define toda aquella acción iniciada por un usuario durante la ejecución de un programa. Cuando el programa detecta un \_\_\_\_\_, se crea un \_\_\_\_\_ designado a realizar una acción relacionada con él.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 6. APLICACIÓN DE LAS ESTRUCTURAS DE ALMACENAMIENTO

En programación es tan importante mantener una organización estricta de los datos provenientes del exterior de nuestro programa, como de los datos definidos internamente en él.

Para tal propósito, las **estructuras de almacenamiento de la información** son la mejor herramienta aliada que nos ofrecen los lenguajes de programación. En ocasiones, veremos que las variables simples capaces de almacenar un único tipo de dato no serán suficientes para conseguir que el programa cumpla sus propósitos.

Las estructuras de almacenamiento nos ofrecen una serie de objetos capaces de almacenar varios datos a la vez e incluso que puedan ser éstos de distinto tipo. Se trata de estructuras destinadas a almacenar bloques de información compleja de una manera estructurada y que permita un acceso sencillo y directo a los datos.

Cuando la información externa a nuestro programa proviene de registros o bases de datos, es una buena praxis almacenarla dentro de nuestro programa con una estructura que imita la original. Para ello, las estructuras de almacenamiento están provistas de toda una serie de características que facilitarán tanto su almacenamiento como su acceso posterior.

Por otra parte, debemos gestionar vectores de elementos (arrays en Java) con un tipo de datos determinado. A veces, este tipo de datos será simple (int, char...); pero, en ocasiones, el vector puede contener elementos que son de un tipo de datos complejo, como por ejemplo objetos de una clase determinada.

También veremos un caso especial de arrays, el de las cadenas de caracteres. En diversos problemas se nos plantean operaciones con cadenas de caracteres, como comparación de cadenas, búsqueda de cadenas dentro de cadenas, ordenación de caracteres, y otras operaciones usuales con las que nos podemos encontrar.

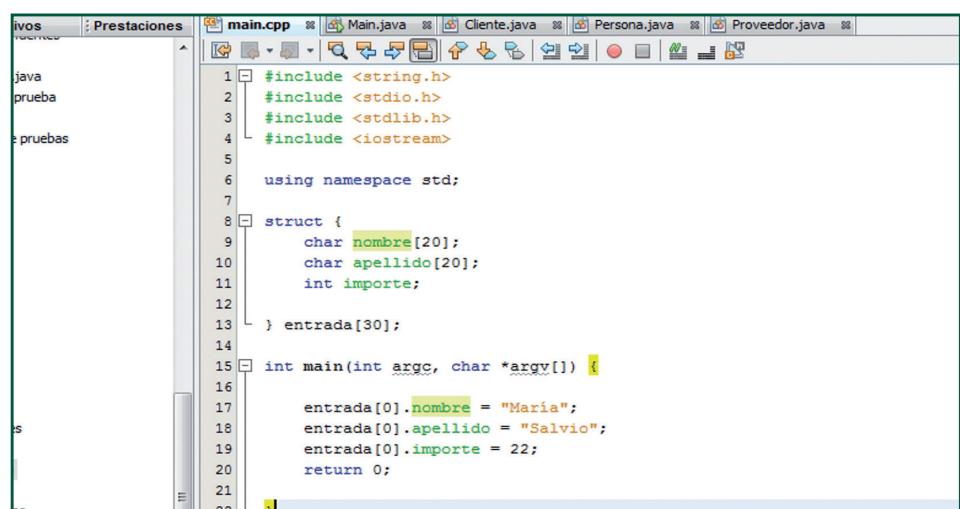
### 6.1 Estructuras

Una **estructura** es un grupo de elementos heterogéneos relacionados de forma conveniente con el programador y el usuario del programa. El resultado es un nuevo tipo de dato que permite una estructura organizativa más compleja que los tipos elementales que ofrecen un lenguaje. Se trata de una combinación de varios tipos de datos, incluyendo otras estructuras que hayamos podido definir previamente.

En el ejemplo de la figura 6.1, el programa empieza definiendo una estructura con la palabra clave **struct** seguida de tres variables sencillas, las cuales representan sus componentes. Seguidamente, encontramos una variable “entrada” que está definida como un vector de 30 posiciones cuyo tipo de datos es esa estructura. Es decir, cada uno de los elementos del vector entrada tendrá en su contenido las tres variables de la estructura.

En el main, podemos ver cómo asignar el valor de un elemento. Para ello, situados con índice 0, se debe dar valor a todo el contenido del elemento; es decir, a todas las variables de la estructura, tal y como se ve en la figura dentro del main.

En lenguajes orientados a objetos como Java, las estructuras se solucionan con clases y objetos, pues las propiedades de los mismos cumplen perfectamente con el cometido de las estructuras.



```

javaprestaciones
main.cpp
1 #include <iostream>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace std;
7
8 struct {
9     char nombre[20];
10    char apellido[20];
11    int importe;
12
13 } entrada[30];
14
15 int main(int argc, char *argv[]) {
16
17     entrada[0].nombre = "Maria";
18     entrada[0].apellido = "Salvio";
19     entrada[0].importe = 22;
20
21     return 0;
22 }

```

**Figura 6.1**  
Definición de una estructura aplicada a dos variables en lenguaje C++.

## 6.2 Creación de arrays

En programación, el término inglés **array** (matriz o vector) define una zona de almacenamiento continuo de datos de un mismo tipo. Desde un punto de vista lógico, una matriz puede entenderse como un conjunto de elementos ordenados en serie. Las matrices son adecuadas para situaciones en las que el acceso a los datos deba realizarse de forma aleatoria e impredecible (Figura 6.2).

Todo vector o matriz se compone de un determinado número de elementos. Cada elemento está referenciado por un **índice** o posición que ocupa dentro del vector.

**Figura 6.2**

Un array donde el índice es la etiqueta y los valores lo que se guarda en el interior.

Existen tres formas de indexar los elementos de una matriz:

- **Indexación base-cero (0).** El primer elemento del vector recibe el índice numérico '0'. En consecuencia, el último elemento del vector recibirá un índice numérico igual al número de elementos totales menos uno. El lenguaje C es un ejemplo típico que utiliza este modo de indexación.
- **Indexación base-uno (1).** En esta forma de indexación, el primer elemento de la matriz tiene el índice '1' y el último tiene el índice igual al número de elementos totales.
- **Indexación base-n (n).** Se trata de un modo de indexación en la que el índice del primer elemento puede ser elegido libremente. En algunos lenguajes de programación se permite que los índices sean negativos e incluso también cadenas de caracteres.

Declaración de arrays en Java:

```
nombre = new <tipo>[dimensión];
Un ejemplo con un tipo simple.
int[] notas;
notas = new int[10];
```

Pero también se puede aplicar a objetos, es decir, a tipos de datos complejos:

```
Persona[] personas = new Persona[10];
```

### 6.3 Inicialización

Como toda otra variable, cuando definimos una matriz en nuestro código, ésta se encuentra vacía. La **inicialización** de una array es el proceso a través del cual definimos el número de elementos que la conformarán y asignamos datos a cada uno de ellos. Existen diferentes métodos para inicializar un array (Figura 6.3), todos ellos el resultado de una colección de datos ordenados de manera correlativa, de la que deberemos escoger el que mejor se ajuste a nuestras necesidades:

- **Inicialización directa.** Consiste en asignar los datos para cada índice de la matriz de modo manual, esto es, asignar manualmente un valor a cada uno de los índices de la matriz.

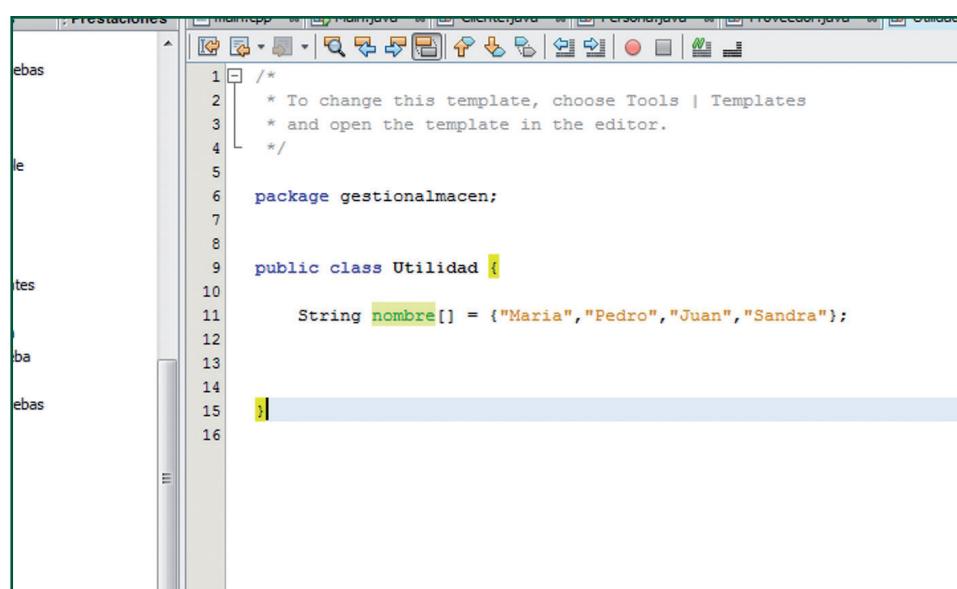
En Java, por ejemplo:

```
<tipo> [] nombreDeTabla = { valor1, valor2, valor3... }
```

- **Inicialización dinámica.** La filosofía de la inicialización dinámica es automatizar el proceso de asignación de datos para cada uno de los índices de la matriz. Este proceso es especialmente útil cuando los datos que debemos almacenar provienen de una fuente externa al programa, como por ejemplo de un documento XML. Gracias a la ayuda de una estructura de repetición (*for, while*) seremos capaces de introducir y ordenar directamente los diferentes datos externos en la matriz.

#### Recuerda

**La mejor manera de recorrer una tabla es la estructura de control for →**  
**for(int=0;i<personas.length;i++)**



The screenshot shows a Java code editor with the following content:

```

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package gestionalmacen;

public class Utilidad {

    String nombre[] = {"Maria", "Pedro", "Juan", "Sandra"};
}

```

The code defines a class named Utilidad with a static array named nombre containing four string elements: "Maria", "Pedro", "Juan", and "Sandra". The array is initialized directly within the class definition.

**Figura 6.3**

Inicialización directa de un vector de String en Java.

En Java, para referirnos a un elemento en concreto de la tabla, lo haremos como en el ejemplo siguiente que imprime el nombre del elemento 3, cuarto en realidad, de la tabla de personas.

```
System.out.println(personas[3].getNombre());
```

Se debe tener en cuenta que el elemento 3, es el de posición 4 dentro de la tabla, pues el índice empieza en 0 , por lo que, en el ejemplo anterior, saldría por consola el nombre de Sandra.

## Recuerda

**La inicialización dinámica de arrays resulta una herramienta valiosísima en cuanto a almacenaje de datos se refiere debido al sistema de automatización que ofrece.**

## 6.4 Arrays multidimensionales

Los **arrays multidimensionales** son unas estructuras de datos capaces de almacenar valores en diferentes niveles. Los arrays que hemos visto hasta ahora almacenan valores en una única dimensión, es decir, con un índice asignado para cada una de sus posiciones y elementos. Los arrays de dos dimensiones (Figura 6.4) guardan sus valores en una estructura similar a las filas y las columnas de una tabla, por ello se requieren dos índices para acceder a cada una de las posiciones de sus datos. En un sentido amplio, podemos entender un **array multidimensional** como un **array** que contiene otro **array** indexado en cada una de sus posiciones (Figura 6.5).

Para referenciar un dato contenido en un array multidimensional deberemos indicar dos posiciones numéricas, la primera hará referencia al índice de la matriz principal y la segunda al índice de la posición del dato de la matriz contenida en él.



**Figura 6.4**

Un array de dos dimensiones necesita dos índices (la fila y la columna).

En Java, la declaración de un array bidimensional del tipo hoja de cálculo, es decir, con filas y columnas, se efectúa de la manera siguiente:

```
<UnTipo> tabla [][] = new <UnTipo> [unaDimensión][];
```

En este caso, sólo se sabe que los elementos serán del tipo UnTipo y que tendrá un número de filas igual a unaDimensión.

Pensemos en un hotel que tenga 4 plantas (0, 1, 2 y 3) y en cada planta tiene 5 habitaciones (de la 0 a la 4). Pequeño, pero útil para la explicación.

En cada habitación podemos tener huéspedes alojados de un tipo de datos determinado. Podríamos guardar enteros, y entonces tendríamos:

```
int hotel[][] = new int [4][];
hotel[0] = new int [5];
hotel[1] = new int [5];
hotel[2] = new int [5];
hotel[3] = new int [5];
```

Y para saber el huésped de la segunda planta primera habitación podemos hacer:

```
habitacionElegida = hotel[1][0];
```



**Figura 6.5**

Un array de tres dimensiones necesita tres índices: fila, columna y piso.

## 6.5 Cadenas de caracteres

Las **cadenas de caracteres** (`String`) nos permiten almacenar conjuntos seriados de caracteres. En ese sentido, podemos entender la mayoría de datos que almacenamos en una variable de tipo `String` como palabras o texto en general. Sin embargo, para el compilador de un lenguaje de programación, un `String` es tratado como una serie de caracteres colocados en serie, a los cuales se les asigna una posición propia dentro de la variable contenedora. Así pues, las cadenas de caracteres son entendidas por el compilador del lenguaje, no como una entidad en sí, sino como un array de caracteres o letras con posiciones individuales para cada una de ellas. Del mismo modo que podemos acceder a un dato indexado dentro de una matriz, podremos también acceder a un carácter concreto dentro de una variable de tipo `String` indicando su posición dentro de ella.

### Las cadenas en Java

#### Declaración

En Java, una cadena de caracteres es un *String*. Su declaración es:

```
String str1;
```

#### Inicialización habitual

Caso más usual:

```
String str1 = "Hola";
```

String inicializado a partir de una cadena de caracteres

```
char t[]="ABC";
```

```
String str1 = new String(t);
```

#### Operaciones (métodos)

En Java, la clase `String` proporciona a los objetos creados con dicha clase, es decir a las variables con ese tipo, operaciones (métodos) de comparación, copia y reemplazo entre otras.

## 6.6 Listas

Una **lista** una estructura de almacenamiento de datos capaz de incrementar o reducir de forma dinámica el número de elementos que contiene. Al mismo tiempo, los elementos contenidos dentro de una **lista** no tienen necesidad de ser del mismo tipo (Figura 6.6). La ventaja principal de una **lista** con respecto a una matriz corriente, es que ésta puede contener tantos tipos de objetos de datos como el programador necesite. La instrucción que define una **lista** dentro del código es **`ArrayList`**. Para inicializar una lista podemos utilizar dos métodos, bien definiéndo-

### Para saber más

De todas las matrices multidimensionales, las de uso más común son las de dos dimensiones. Sin embargo, a medida que dotemos un array de mayores dimensiones, podremos conseguir efectos más complejos para nuestro programa, ya sea para almacenar datos o ya sea para representar objetos gráficos por pantalla.



Para ampliar este tema, puedes ver el videotutorial *Creación de clases, objetos y ficheros en Java, que encontrarás en el Campus online*.

la como vacía y, por tanto, sin número inicial de elementos, o bien definiendo un número determinado de posiciones iniciales. Ambos sistemas son completamente válidos, pero es recomendable que, para economizar memoria, si conocemos de antemano el número de elementos mínimos que vayamos a necesitar, lo definamos al declarar la **lista**.

**Figura 6.6**

En las listas, cada elemento puede ser de un tipo diferente, incluyendo objetos.

		0	Maria	
		1	53	
		2	456,23	
	3			

En Java, la gestión de listas comprende los siguientes aspectos:

#### • Declaración

En Java, para poder utilizar una lista, no se puede crear un objeto de la clase *List*, ya que es abstracta y, por lo tanto, es necesario crear un objeto de *ArrayList*.

```
ArrayList losTelefonos = new ArrayList();
```

En este caso, *losTelefonos* será un array cualquier tipo de elemento. Pero si, por ejemplo, tenemos una clase *Persona* y queremos crear una lista de personas, debemos insertar la clase a la que nos referimos entre < >. De esta forma:

```
ArrayList<Persona> unasPersonas = new ArrayList<Persona>();
```

#### • Inicialización habitual

En cualquier caso, se emplea el método *add*. Por ejemplo:

```
Persona laPersona = new Persona();
unasPersonas.add(laPersona);
```

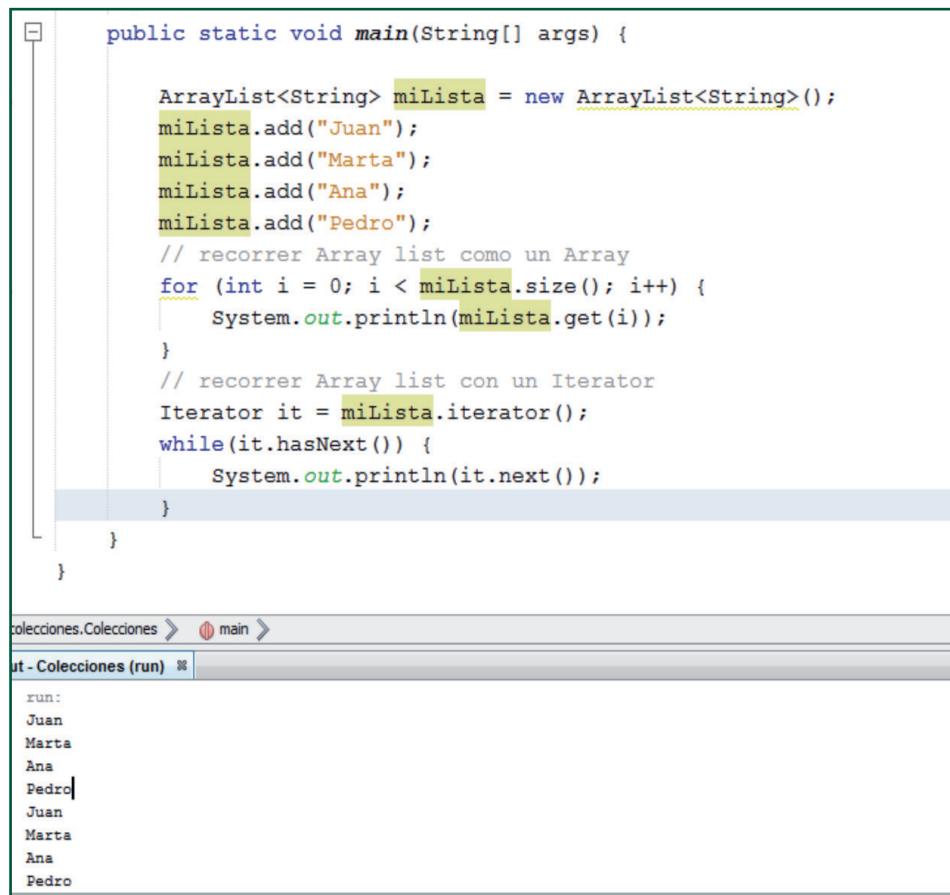
#### • Operaciones (métodos)

En Java, un objeto del tipo *List*, aparte de la operación añadir descrita, tiene operaciones de borrado, cuenta del número de elementos, búsqueda de un elemento, etc. Por otra parte, la lista proporciona la propiedad *Iterator* que se utiliza para recorrer la lista.

## 6.7 Colecciones

Una colección, llamada comúnmente en inglés **Collection**, es un sistema, implementado en Java mediante clases, que permite agrupar una serie de objetos de modo que se puede recorrer o iterar, y del cual conocemos su tamaño (Figura 6.7). Las operaciones básicas de una colección son las siguientes:

- **add(T)**. Añade un elemento.
- **Iterator()**. Obtiene un “iterador” que permite recorrer la colección visitando cada elemento una vez.
- **size()**. Obtiene la cantidad de elementos almacenados dentro de una colección.
- **contains(t)**. Devuelve un valor booleano (cierto o falso) según si el elemento entre paréntesis (t) se encuentra dentro de la colección o no.



The screenshot shows an IDE interface with two main sections. The top section is a code editor containing Java code for creating an ArrayList of strings and printing its elements using both an array-style loop and an iterator. The bottom section is a terminal window showing the execution of the code and its output.

```

public static void main(String[] args) {
    ArrayList<String> miLista = new ArrayList<String>();
    miLista.add("Juan");
    miLista.add("Marta");
    miLista.add("Ana");
    miLista.add("Pedro");
    // recorrer Array list como un Array
    for (int i = 0; i < miLista.size(); i++) {
        System.out.println(miLista.get(i));
    }
    // recorrer Array list con un Iterator
    Iterator it = miLista.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}

```

colecciones.Colecciones > main >

Output - Colecciones (run) >

```

run:
Juan
Marta
Ana
Pedro
Juan
Marta
Ana
Pedro

```

**Figura 6.7**

Ejemplo de uso de un ArrayList.

La figura 6.7 muestra cómo crear un ArrayList de String y cómo usar los métodos size() y get() para acceder a los elementos contenidos en la lista. También muestra cómo se puede acceder usando un iterator.

La característica principal de un objeto **Collection** es la de poder ser recorrido. Si bien en este punto del proceso no está definido un orden, la única manera de definirlo es usando una instrucción iteradora de repetición, mediante el método *iterator()*. Un iterador es un objeto que recorre la extensión de la colección y nos permite obtener todos los objetos contenidos en ella, invocando progresivamente un método llamado *next()*. Si la colección es modificable, podremos eliminar un objeto durante el recorrido mediante el método llamado *remove()* del iterador.

En Java la gestión de colecciones se efectúa a través de las clases heredadas de *Collection*, como son *ArrayList*, *Vector*...

## Resumen

Una **estructura** es un grupo de elementos relacionados entre sí, que es el resultado de un nuevo tipo de dato considerablemente más complejo que los que hemos utilizado hasta ahora. Se trata de una combinación de varios tipos de datos, incluyendo otras estructuras que hayamos definido previamente. Se utilizan en lenguajes que no incorporan la POO, ya que las clases se pueden considerar una evolución de las estructuras.

Un **array** define una zona de almacenamiento continuo de datos de un mismo tipo. Desde un punto de vista lógico, una matriz puede entenderse como un conjunto de elementos ordenados en serie. Las matrices son adecuadas para situaciones en las que el acceso a los datos deba realizarse de forma aleatoria e impredecible.

La **inicialización** de una array es el proceso a través del cual definimos el número de elementos que la conformarán y asignamos datos a cada uno de ellos.

Los **arrays multidimensionales** son estructuras de datos capaces de almacenar valores en diferentes niveles. En ellos se almacenan los valores en una estructura similar a las filas y las columnas de una tabla.

Las **cadenas de caracteres** (String) nos permiten almacenar conjuntos seriados de caracteres. Para el compilador de un lenguaje de programación, un string es tratada como una serie de caracteres colocados en serie con una posición propia para cada uno de ellos. Así pues, las cadenas de caracteres son entendidas como un array de caracteres o letras.

Una **lista** es una estructura de almacenamiento de datos capaz de incrementar o reducir de forma dinámica el número de elementos que contiene. Las ventajas de una **lista** frente a un array común son la capacidad de almacenar elementos de distinto tipo en su estructura y las modificaciones en el número de elementos.

Una **Collection** es el concepto base del que derivan la mayoría de las estructuras de almacenamiento de información. Su característica principal es la de poder obtener datos de él y modificarlos a la vez que se recorre su extensión.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Una estructura es un grupo de elementos homogéneos relacionados de forma conveniente con el programador y el usuario del programa.
2. En lenguajes orientados a objetos, como Java, las estructuras se solucionan con clases y objetos, pues las propiedades de los mismos cumplen perfectamente con el cometido de las estructuras.
3. En programación, el término inglés *array* (matriz o vector) define una zona de almacenamiento continuo de datos de un mismo tipo.
4. Todo vector o matriz se compone de un determinado número de elementos. Cada elemento está referenciado por un índice o posición que ocupa dentro del vector.
5. Indexación base-n (n) es un modo de indexación en la que el índice del primer elemento puede ser elegido libremente. En ningún lenguaje de programación se permite que los índices sean negativos.
6. La inicialización directa consiste en asignar automáticamente un valor a cada uno de los índices de la matriz.
7. Una colección, llamada comúnmente en inglés *Collection*, es un sistema, implementado en Java mediante clases, que permite agrupar una serie de objetos de modo que se pueda recorrer o iterar, y cuyo tamaño conocemos.

**Completa las siguientes afirmaciones:**

8. Indexación base-cero (0). El primer elemento del vector recibe el \_\_\_\_\_ numérico '0'. En consecuencia, el último elemento del vector recibirá un índice numérico igual al número de elementos totales menos \_\_\_\_\_.
9. La filosofía de la inicialización \_\_\_\_\_ es automatizar el proceso de asignación de datos para cada uno de los índices de la \_\_\_\_\_. Este proceso es especialmente útil cuando los datos que debemos almacenar pro-

vienen de una fuente externa al programa, como por ejemplo de un documento \_\_\_\_\_.

10. Los arrays multidimensionales son unas estructuras de datos capaces de almacenar \_\_\_\_\_ en diferentes niveles. En un sentido amplio, podemos entender un array multidimensional como un array que contiene otro array \_\_\_\_\_ en cada una de sus posiciones.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 7. UTILIZACIÓN AVANZADA DE CLASES

El estudio de las clases y su distintos usos y posibilidades dentro de la **programación orientada a objetos (POO)** es muy amplio. Ya hemos dado anteriormente una explicación global a estos conceptos.

El objetivo de esta unidad es la de ofrecer una visión más específica sobre sus características. Veremos cómo la **herencia** supone un concepto mucho más amplio que el simple hecho de compartir características entre **clases** y cómo puede ofrecernos un mayor control de nuestros programas.

Analizaremos cómo debemos entender y dotar de un sentido más específico el proceso de definición de **clases** y cómo puede ayudarnos a simplificar la estructura de nuestros códigos.

Veremos también la importancia del **anidado y empaquetado de clases** creando grupos organizados de objetos capaces de realizar tanto acciones compartidas como específicas, y cómo, a partir de una misma función, pueden derivarse otras funciones con unas modificaciones determinadas. Explicaremos también el uso de clases y de cómo estas pretenden representar objetos y conceptos de la vida cotidiana.

Avanzaremos en el concepto de polimorfismo y en su utilización como herramienta para desarrollar funciones que, aunque tengan el mismo identificador, su implementación es diferente según los parámetros de entrada.

La aplicación de estos conceptos, siguiendo el camino empezado en las unidades anteriores, la llevaremos a cabo a través del lenguaje de programación Java.

Las instrucciones del lenguaje, y sobre todo, las librerías de objetos que podemos utilizar, hacen del mismo una potente solución para la programación orientada a objetos, para el cometido de esta unidad: el uso avanzado de clases, la herencia y el polimorfismo. Java no tiene herencia múltiple, sin embargo, este hecho no ocasiona ningún problema para transformar cualquier modelo de objetos en una verdadera implementación totalmente funcional, ya que podemos usar las interfaces, que obligan a un comportamiento y apoyan la transformación del modelo indicado.

### 7.1 Composición de clases

La **composición de clases**, también conocida como **relación asociativa**, es la relación que se establece entre dos objetos que tienen una comunicación persis-

tente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones. La composición de clases implica que uno de los dos objetos involucrados está **compuesto** por el otro. Desde el punto de vista práctico, es posible reconocer asociatividad entre dos clases A y B si la proposición "A **tiene un** B" es verdadera. Por ejemplo: "un ordenador tiene un teclado" es verdadero. Por tanto, un objeto de tipo Ordenador tiene una relación de **composición** con un objeto de tipo Teclado (Figura 7.1). Los objetos que componen una clase contenedora deben existir desde el inicio del programa. No hay posibilidad de que una **clase contenadora** pueda existir sin alguno de sus objetos componentes, razón por la que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.

Siguiendo la premisa de dependencia entre dos clases, es fácil confundir el concepto de **composición** con el concepto de **herencia**, sin embargo existe una notable diferencia entre ellos. Mientras que la **composición** implica que un objeto contiene al otro, la herencia implica que un objeto deriva del otro. Es decir, según la **composición** un ordenador posee un teclado pero no es un teclado.



**Figura 7.1**

Equipo como la composición de objetos (torre, pantalla, teclado y ratón).

En la unidad anterior, concretamente en el estudio de AWT, ya hemos visto la implementación concreta de la composición en Java. Un contenedor Frame que contiene por ejemplo una etiqueta, un cuadro de texto y un botón, que se han creado como propiedades de la ventana principal, es una implementación de composición.

Estos componentes concretos (no su clase general) no tienen razón de existir sin estar contenidos en el objeto principal. No podemos crear objetos de ellos en otras clases, son lo que son dentro de la ventana y no fuera.

```
public class AWTCounter extends java.awt.Frame implements ActionListener {  
  
    private Label lblCount;  
    private TextField tfCount;  
    private Button btnCount;  
    ...  
}
```

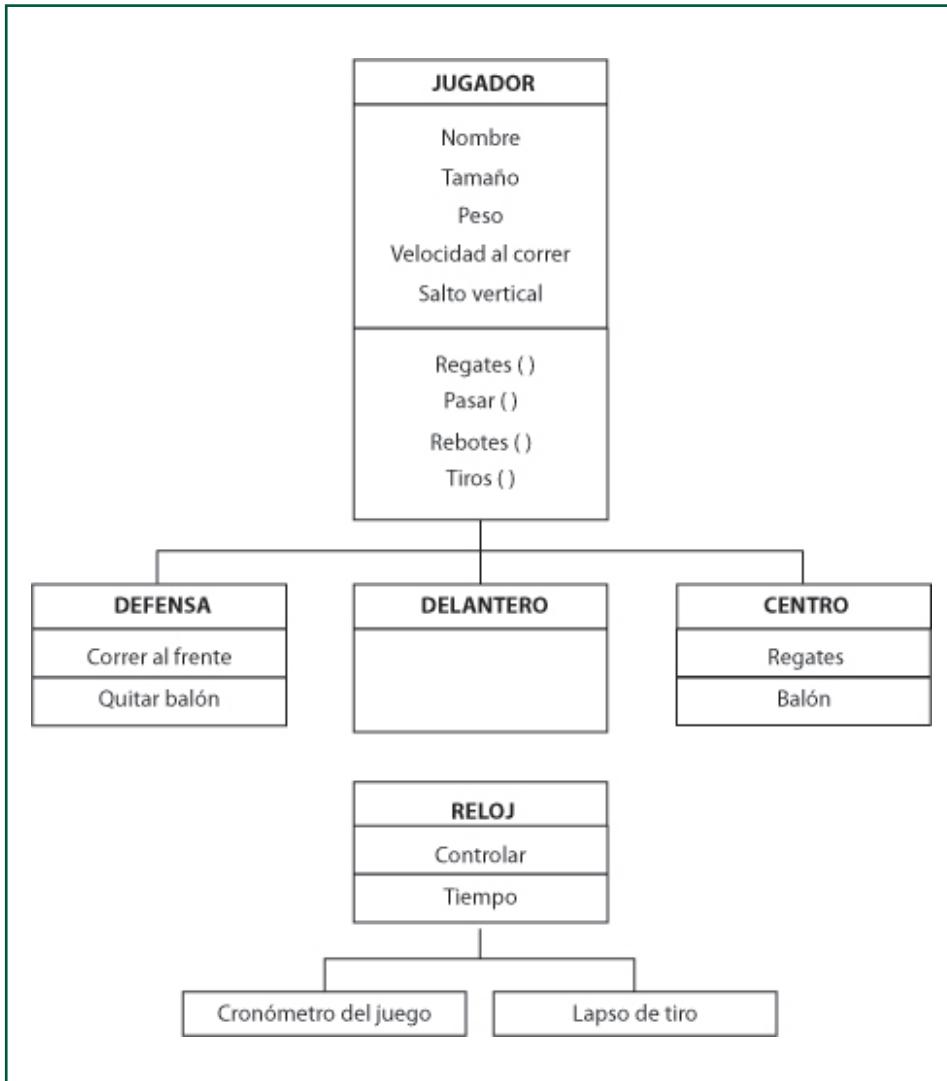
## 7.2 Herencia

El concepto de herencia hace referencia a la relación existente entre dos objetos o clases, en la que una deriva de la otra. Mientras que la composición simplemente indica la existencia de una clase dentro de otra, y, por lo tanto, sólo se puede usar el objeto "como es (as is)", la herencia actúa como cesión de propiedades entre clases y, por lo tanto, se pueden redefinir las clases heredadas a partir de las superiores. Si tomamos como ejemplo un objeto llamado *Persona*, en el que se definen una serie de propiedades como nombre, edad, sexo, etc., y otro llamado *Empleado* que deriva de él, se entiende que el objeto *Empleado* poseerá, además de sus propias características, todas las del objeto *Persona*. En ese sentido, podemos pues decir que un *Empleado* es una *Persona*, premisa que no se cumple en la composición entre dos clases. Toda clase que deriva de otra es llamada *subclase*.

La herencia nos permite crear una estructura jerárquica de clases cada vez más especializada (Figura 7.2). La mayor ventaja que ofrece este proceso es que cuando se quiere especializar una clase existente no es necesario empezar desde cero, sino que basta con hacer que ésta herede las propiedades y los métodos de otra más global. Como resultado, es posible construir bibliotecas de clases que ofrecen una base que podrá ser especializada según lo creamos conveniente.

En diversos lenguajes de programación existe el concepto de herencia múltiple, que significa que una clase puede heredar de varias a la vez, adoptando de este modo propiedades y atributos pertenecientes de varias superclases.

En Java una clase sólo puede heredar de una a la vez, sin embargo, a su vez, puede implementar una o varias interfaces, hecho que dota al lenguaje de la posibilidad de implementar cualquier modelo de clases, por complejo que sea.

**Figura 7.2**

Herencia de clases de jugadores de fútbol.

Como hemos visto en unidades anteriores, la sintaxis en Java para la herencia es:

```
class Profesor extends Persona
```

donde Profesor hereda las propiedades y métodos de Persona.

Por otra parte, podemos tener otras clases como Alumno y PAS (personal no docente) que heredan de Persona. Sin embargo, nos interesa que Profesor y PAS hereden también de la clase Trabajador, y en Java esto no es posible.

Sin embargo, podemos pensar en por qué deben heredar de Trabajador y, realizando un análisis, llegamos a la conclusión de que están obligados a fichar y a cobrar la nómina.

## Recuerda

**Existe una notable diferencia entre la composición y la herencia, pues mientras que la composición es un objeto que contiene al otro, la herencia es un objeto deriva del otro.**

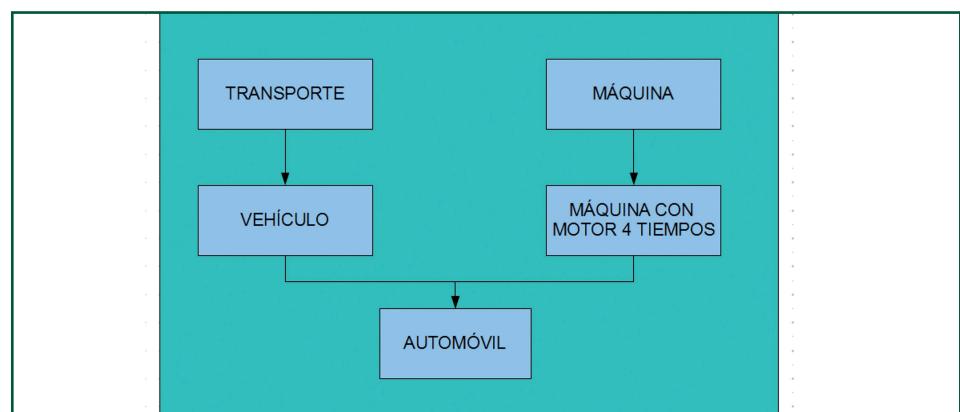
En tal situación, se puede pensar en la clase Trabajador como una interfaz:

Interfaz Trabajador{

```
    public void fichar(Persona p){  
    }  
    public void cobrar(Persona p){  
    }  
}
```

Y en tal situación, Profesor y PAS implementan esa interfaz. Es decir, están obligados a implementar sus métodos, pero cada uno fichará y cobrará según un algoritmo específico.

```
public class Profesor extends Persona implements Trabajador{  
    public void fichar(Persona p){  
        // Código x ....  
    }  
  
    public void cobrar(Persona p){  
        // Código Y  
    }  
}
```



## 7.3 Superclases y subclases

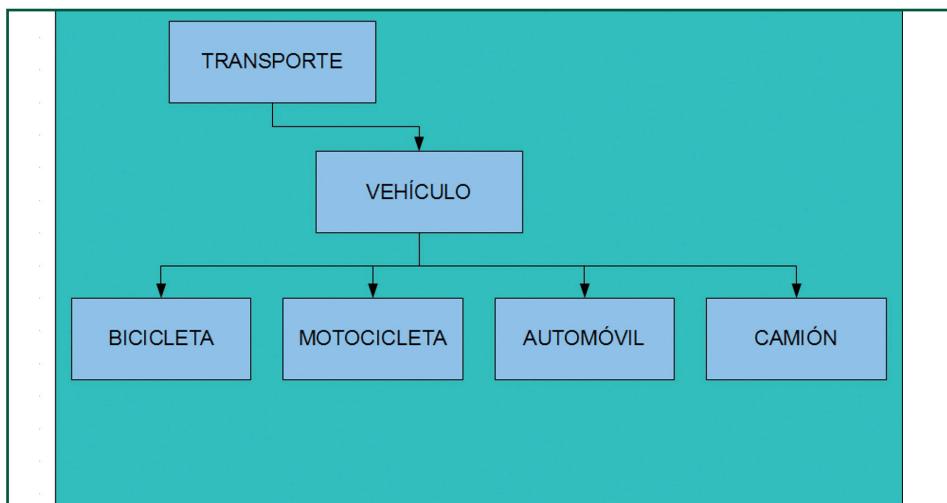
**Figura 7.3**

Jerarquía de clases con herencia múltiple.

**Superclase** y **subclase** son los nombres que reciben las clases generales y sus clases derivadas respectivamente. Una subclase es un objeto que deriva de una clase más general

y que, como tal, adquiere todas las propiedades especificadas en ella. Siguiendo el ejemplo anterior en el que proponíamos dos clases llamadas *Persona* y *Empleado*, podemos definir que *Persona* se refiere a la clase madre y *Empleado* a una clase derivada; es decir, *Persona* es la superclase de *Empleado* y *Empleado* es una **subclase** de *Persona*.

La creación de **subclases** nos permite crear nuevas clases personalizadas y más específicas tomando los elementos necesarios de una clase ya existente. De este modo, nos ahorramos tener que volver a escribir todas las propiedades, métodos y eventos que ya posee la clase madre. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, e incluso ocultarlas para que no se pueda acceder desde el exterior. En programación, cuando una clase deriva de otra, se dice que la extiende, y la instrucción utilizada para indicarlo es **extends** seguido del nombre de la clase madre (Figura 7.4).



**Figura 7.4**  
Vehículo es superclase de motocicleta y bicicleta y, a su vez, subclase de transporte.

## 7.4 Clases y métodos abstractos y finales

**Abstracto** (en inglés *abstract*) y **final** (en inglés *final*) son dos tipos específicos de declaración de clases y métodos de clase. En cuanto a las clases, la especificación *abstract* hace referencia a clases que no permiten su instanciación directa en el código, pero sí en cambio que otras clases extiendan de ellas. Los métodos declarados como *abstract* son funciones definidas destinadas a realizar acciones directas. La diferencia básica entre un método o función *abstracta* y otro común es su estructura o modo de definición. Mientras que en una función común se especifican las acciones a realizar con un bloque de código delimitado entre corchetes, en una función *abstracta* sólo se deja preparado el método, pero sin código, para que sea implementada en las subclases. Una clase debe declararse *abstracta* sólo cuando contenga métodos

## Para saber más

**Una clase abstracta puede tener definidos datos y métodos estáticos. Para utilizar una de estas propiedades, sólo hay que referenciar la clase a la que pertenecen. Por ejemplo, ClaseAbstracta.metodoEstatico(), como se haría con cualquier otra clase.**

**Figura 7.5**

Ejemplo de clase abstracta con método abstracto.

abstractos definidos en su estructura (Figura 7.5). Este mecanismo es similar a las funciones virtuales puras del C++ que hacen que la clase sea abstracta.

La especificación *final* se usa para definir una clase de la que no podrá derivar ninguna otra. Se trata de un parámetro destinado a indicar que una clase representa el final de la jerarquía parcial o total de una biblioteca de clases.

```

1  abstract class FiguraGeometrica {
2      ...
3      abstract void dibujar();
4      ...
5  }
6
7  class Circulo extends FiguraGeometrica {
8      ...
9      void dibujar(){
10         //código para dibujar círculo
11     }
12 }
13

```

## 7.5 Sobreescritura de métodos

Cuando definimos una subclase, ésta hereda todas las propiedades y métodos de su clase madre de manera automática. Sin embargo, en ocasiones, puede ocurrir que una subclase esté destinada a realizar acciones definidas en su clase madre pero con ciertas variaciones. Dicha situación resulta de una interferencia entre ambas funciones, y para evitarlo debe utilizarse el parámetro **override** (sobreescribir) para aquella función definida en la subclase (Figura 7.6). El parámetro **override** indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la subclase las que prevalecerán.

Pongamos como ejemplo una clase principal llamada *Animal*, en la cual se definen las acciones básicas que realizan todos los animales sea cual sea su especie (comer, beber, correr, etc.). Los métodos definidos en la superclase son heredados por todas las subclases que de ella se deriven, pero sólo definen acciones genéricas, es decir, se sobreentiende que todos los animales beben agua, aunque no todos lo hacen de la misma manera. Un perro o un gato utilizan un mismo método, pero para el caso de un elefante, por ejemplo, es distinto. Así pues, en la definición del método *beber* de la clase *Elefante* utilizaríamos el parámetro *override* para definir las modificaciones necesarias.

```

1  @Override
2  public int hashCode() {
3      int hash = 0;
4      hash += (codiActivitat != null codiActivitat.hashCode()):0;
5      return hash;
6  }
7

```

**Figura 7.6**

Ejemplo de subclase donde los métodos que se reescriben se anotan con `@override`.

En Java, como se ha explicado, definir que una clase hereda de otra se realiza en la declaración con la palabra reservada *extends*:

```

classe Profesor extends Persona{
}

```

Una clase declarada así hace que, aunque no estén en el código, todas las propiedades y métodos de Persona estén incluidas en Profesor. Imaginemos un método de Persona que muestra el nombre de la persona a partir de su NIF (por ejemplo, para mostrarlo en una ventana de una aplicación). La clase *Profesor* puede tener una propiedad añadida con el departamento y nos interesa que en dicha aplicación también se muestre. En tal situación, el método de Persona sería simplemente mostrar el nombre y el método de Profesor reescribiría el de Persona, llamando a éste mediante la sentencia `super.metodo (parámetros)` y, además, mostrando el departamento.

## 7.6 Constructores y herencia

Como hemos definido anteriormente, los **constructores** tiene el mismo nombre que las clases. El **constructor** funciona al mismo tiempo como función principal e identificador para una clase. Recordemos que, para instanciar una clase en nuestro código, sólo necesitamos definir un dato para almacenarla y llamar a su **método constructor** para crearla. De ese modo, podemos importar en nuestro código cualquier tipo de clase, sin embargo este proceso no siempre tiene sentido. Siguiendo el ejemplo de una clase madre llamada *Animal*, nos damos cuenta de que, al instanciarla en el código, poco podemos hacer con ella, ya que, por norma general, los datos y funciones que contiene son genéricos, es decir, poco específicos. Es aquí donde toma importancia la **herencia**. Ésta sirve no sólo para transmitir propiedades entre clases, sino también para dotar de sentido a las acciones definidas en ellas. Que un animal puede beber y comer es evidente, pero no es representable si no asociamos dichas acciones a algún animal en concreto. Así pues, gracias a la herencia, al crear una clase llamada *Perro*, que deriva de otra llamada *Animal*, lo que en realidad estamos haciendo es dotar de sentido a las

### Recuerda

**Para evitar la interferencia entre las funciones definidas en las subclases y las superclases, deberemos utilizar el término `override` para indicar las modificaciones que incluirá la función que imperará de las dos.**

acciones definidas en la clase *Animal*. La estructura de una subclase es idéntica a la de cualquier clase, el único factor que varía entre los distintos lenguajes de programación es el modo de definir de qué clase hereda las propiedades.

En Java, una clase hereda de la superclase todas las propiedades y todos los métodos excepto los constructores.

Como hemos aprendido, en Java podemos omitir un constructor. Sin embargo, esto no quiere decir que no exista, sino que Java lo creará en tiempo de ejecución, aunque no haga nada explícito.

Si en la superclase se añade un constructor en el código, en su diseño, la subclase no heredará el mismo, pero sí se puede codificar uno que llame al de la superclase con la palabra reservada *super*.

## 7.7 Acceso a métodos de la superclase

Tal y como estipulan las directrices de la herencia entre clases, los métodos definidos en una clase madre o superclase, pueden ser utilizados por todas las clases que de ella se deriven. Para ello, el requisito principal es que toda la clase que queramos definir como global para todas las subclases sea de tipo público (**public**). Con la especificación *public* podremos acceder a una función desde puntos externos de la clase en la que ha sido definida. Para el caso de las clases, la filosofía es la misma, sólo que conlleva la característica de que la función no sólo puede ser accedida desde una subclase si no que, además, dicha función pasa a pertenecerle. Al definir una clase llana, si introducimos una función pública en ella, ésta podrá ser invocada sólo al instanciar la clase en el código. Sin embargo, tratándose de una subclase que hereda de una clase más general, la instancia ya no será necesaria, ya que la función pasará a formar parte activa de ella. Dicho proceso puede repetirse con todas las subsiguientes clases que definamos a partir de ella, proporcionándonos una estructura de organización sólida que favorecerá el control de nuestros códigos. Así pues, tomando los ejemplos propuestos anteriormente, acciones como *correr*, *beber*, *comer* o *mover*, definidos en una clase llamada *Animal*, deberán ser públicos para poder acceder desde todas sus subclases.

En Java, tal y como se ha mencionado con los constructores, se puede reescribir un método en la subclase. Dicho método puede reescribirse totalmente, o bien, aprovecharse de lo ya implementado en el método de la clase de la cual hereda llamándolo. Esta llamada se efectuará con la palabra reservada *super* (*super.metodoY(xxx)*, donde *metodoY* es el método de la superclase a la que necesitamos llamar) (Figura 7.7).

```

1  public class Proveedor extends Persona {
2      public Double importeCompras = 0.;
3      public static Double importeTotal = 0.;
4      public int a;
5
6      public Proveedor(String nombre, String telefono) {
7          super(nombre, telefono);
8      }
9
10     public Proveedor(String nif){
11         super(nif);
12     }
13 }
14

```

**Figura 7.7**

Ejemplo de constructor con llamada al de la superclase con la palabra reservada "super".

## 7.8 Polimorfismo

En programación orientada a objetos, en ortodoxia, el **polimorfismo** se refiere a la capacidad que poseen las clases derivadas de una antecesora (subclases) para utilizar una misma acción de forma distinta. Tomemos como ejemplo dos clases llamadas *Pez* y *Ave* derivadas de la superclase *Animal*. La clase *Animal* tiene definido el método abstracto *mover*, el cual se implementará de forma distinta en cada una de las subclases, ya que peces y aves se mueven de forma diferente (Figura 7.8). El concepto de **polimorfismo** puede aplicarse tanto a funciones como a tipos de datos.

```

1  #include<iostream>
2  using namespace std;
3
4  class figura {
5  private:
6      float base;
7      float altura;
8  public:
9      void captura();
10     virtual unsigned float perimetro()=0;
11     virtual unsigned float area()=0;
12 };
13 class rectangulo: public figura {
14 public:
15     void imprime();
16     unsigned float perimetro(){return 2*(base+altura);}
17     unsigned float area(){return base*altura;}
18 };

```

**Figura 7.8**

Ejemplo de cómo diferentes clases definidas en lenguaje C++ usan de manera distinta los atributos (datos) y métodos (funciones) definidos en su clase madre llamada figura.

Existen dos tipos de **polimorfismo**:

- **Polimorfismo estático.** Cuando disponemos de un mismo método de varias formas, es en el tiempo de compilación cuando se decide cuál de ellos se usará. En este caso será el número y los tipos de parámetros los factores que permitirán al compilador seleccionar el método.
- **Polimorfismo dinámico.** En una jerarquía de clases, una referencia a un objeto de una clase puede serlo también a otras subclases, por lo que en el tiempo de ejecución se ejecutará un método u otro según a qué tipo de objeto este refiriéndose.

En la teoría ortodoxa de la programación orientada a objetos, el **polimorfismo** es la capacidad de que varios métodos tengan el mismo identificador pero implementen funciones diferentes.

En la práctica, en los lenguajes de programación, esto se consigue haciendo que los parámetros de entrada sean diferentes. El valor devuelto por la función puede ser igual o diferente, pero esto no hará que sea polimórfico.

Java soporta esta definición y esta característica es usada habitualmente. Por ejemplo, si pensamos en la función `System.out.println(zzz)`, es posible que no hayamos caído en la cuenta de que es polimórfica, y lo podemos comprobar yendo a su definición: veremos que se trata de varias funciones. Al usarla, le hemos pasado parámetros diferentes sin darnos cuenta, un string normalmente, pero también un Integer, e incluso le podríamos pasar cualquier tipo de objeto (no se puede asegurar que imprimiría). Al pasarle diferentes tipos de datos, realmente se han ejecutado funciones diferentes, que se llaman igual, que tienen el mismo identificador, pero que están implementadas de forma diferente, aunque al final impriman lo que le hemos pasado, necesariamente son diferentes.

Muchas veces, se habla también de polimorfismo referido a la sobrecarga de métodos, es decir, a cuando reescribimos un método de una superclase en la clase que hereda. Esto también está aceptado, aunque no se desarrollará en este apartado porque ya está descrito en apartados anteriores.

## Resumen

La composición de clases, también conocida como *relación asociativa*, es la relación que se establece entre dos objetos que tienen una comunicación persistente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones.

La composición de clases o relación asociativa entre clases es una relación de pertenencia de un objeto a otro. Aunque el objeto que pertenece al otro puede crearse sin él, no tiene sentido por sí solo en el conjunto. Por ejemplo, un teclado y el ordenador.

En cambio, la herencia es una relación donde una subclase deriva de una superclase, obteniendo las propiedades y los métodos de ella. Por ejemplo, profesor puede heredar de la superclase persona.

Una superclase puede ser abstracta, es decir, no podrá instanciarse y sólo podrá heredarse. En este sentido, sus métodos pueden ser abstractos, con lo que sólo estarán declarados en la superclase y su implementación debe codificarse obligatoriamente en cada subclase. La clase de la cual no se puede heredar debe declararse como final.

Las subclases heredan las propiedades y métodos de la superclase, pudiendo sobreescribir los métodos de la superclase, a los cuales también se puede acceder mediante el uso de la palabra reservada super. Los métodos constructores no se heredan.

El concepto de polimorfismo es un término general de la programación orientada a objetos que corresponde a la posibilidad de que diversos métodos se denominen igual, pero para ello deben cambiar algo en los parámetros de llamada. Este concepto se extiende cuando se habla de herencia porque un método heredado se puede reescribir, aún sin cambiar los parámetros.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. La composición de clases, también conocida como relación asociativa, es la relación que se establece entre dos objetos que tienen una comunicación persistente, es decir, que existe una relación de dependencia para que puedan llevar a cabo sus respectivas funciones.
2. En diversos lenguajes de programación existe el concepto de herencia múltiple, que significa que una clase puede heredar de varias a la vez, adoptando, de este modo, propiedades y atributos pertenecientes de varias superclases.
3. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, pero no podremos ocultarlas.
4. El parámetro *override* indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la clase madre las que prevalecerán.
5. La composición implica que un objeto contiene al otro, y la herencia implica que un objeto deriva del otro.
6. En cuanto a las clases, la especificación *abstract* hace referencia a clases que permiten su instanciación directa en el código, pero no que otras clases extiendan de ellas.
7. En programación orientada a objetos, en ortodoxia, el polimorfismo se refiere a la capacidad que poseen las clases derivadas de una antecesora (subclases) para utilizar una misma acción de forma distinta.

**Completa las siguientes afirmaciones:**

8. Siguiendo la premisa de dependencia entre dos clases, es fácil confundir el concepto de \_\_\_\_\_ con el concepto de \_\_\_\_\_, sin embargo existe una notable diferencia entre ellos. Mientras que la \_\_\_\_\_ implica que un objeto contiene al otro, la \_\_\_\_\_ implica que un objeto deriva del otro.

9. En Java una clase solo puede heredar de una a la vez; sin embargo, a su vez, puede implementar una o varias \_\_\_\_\_, hecho que dota al lenguaje de la posibilidad de implementar cualquier modelo de \_\_\_\_\_, por complejo que sea.
10. Mientras que en una función común se especifican las acciones a realizar con un bloque de código delimitado entre \_\_\_\_\_, en una función abstracta solo se deja preparado el método, pero sin código, para que sea implementada en las \_\_\_\_\_.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 8. MANTENIMIENTO DE LA PERSISTENCIA DE LOS OBJETOS

Una **base de datos** (BBDD) es un conjunto de datos almacenados y relacionados entre sí, con dos características básicas: la información es persistente y el acceso a la información debe ser eficiente (rápido).

Una **BBDD Relacional** es un conjunto de relaciones, término que en las bases de datos se podría tomar como sinónimo de tabla, formada por un esquema y un cuerpo descritos en términos de dominios, atributos y asociaciones. Se trata de estructuras autodescriptivas por medio de sus tablas de relación de datos. En una BBDD se almacenan tanto elementos de datos (**tuplas**), como las relaciones existentes entre ellos, por medio de atributos asignados. A diferencia de una base de datos común, en una **Base de Datos Orientada a Objetos** (OODB - *Object-oriented Database*), los elementos de datos son tratados como objetos. Los datos contenidos en ellas se encuentran representadas como clases. Una BDOO se rige por un método sistemático de representación y relación.

Las bases de datos orientados a objetos se propusieron con la idea de satisfacer las necesidades de aplicaciones más complejas. El enfoque orientado a objetos ofrece la flexibilidad para cumplir con algunos de estos requerimientos al no estar limitado por los tipos de datos y los lenguajes de consulta disponibles en los sistemas de bases de datos tradicionales. Como cualquier base de datos programable, una base de Datos Orientada a Objetos (BDOO) proporciona un ambiente para el desarrollo de aplicaciones y un depósito persistente listo para su explotación. Una BDOO almacena y manipula información que puede ser digitalizada (presentada) como objetos, además de un acceso ágil a los datos y una gran capacidad de manipulación para trabajar con ellos.

Otro motivo para la creación de las bases de datos orientadas a objetos es el creciente uso de los lenguajes orientados a objetos para desarrollar aplicaciones. Las bases de datos se han convertido en piezas fundamentales de muchos sistemas de información y las bases de datos tradicionales son difíciles de utilizar cuando las aplicaciones que acceden a ellas están escritas en un lenguaje de programación orientado a objetos como C++, Smalltalk o Java. Las bases de datos orientadas a objetos se han diseñado para que se puedan integrar directamente con aplicaciones desarrolladas con lenguajes orientados a objetos, habiendo adoptado muchos de los conceptos de estos lenguajes como son la herencia y el encapsulamiento.

## 8.1 Bases de datos orientadas a objetos

En una **base de datos orientada a objetos**, la información se representa mediante objetos, como los que hay presentes en la programación orientada a objetos. Cuando se integran las características de una base de datos con las de un lenguaje de programación orientado a objetos, el resultado es un **sistema gestor de base de datos orientada a objetos** (ODBMS, *object database management system*) (Figura 8.1). Un ODBMS hace posible que los objetos de la base de datos aparezcan como objetos de un lenguaje de programación en uno o más lenguajes de programación a los que dé soporte. Un **ODBMS** recibe las características de los lenguajes como el tratamiento de datos persistentes, control de concurrencia, recuperación de datos, consultas asociativas y otras capacidades. Los principales puntos que se aplican a una base de datos orientada a objetos (BDOO) son los siguientes:

- **Identidad de objetos.** Identificación única para cada uno de los datos.
- **Constructores de tipos.** Permiten la creación de objetos complejos a partir de otros.
- **Encapsulamiento.** Ocultación de las propiedades.
- **Compatibilidad con diferentes lenguajes de programación.**
- **Jerarquías de tipos y herencia.** Clases ordenadas según su función y especialización.
- **Manejo de objetos complejos.** Objetos que requieren un área de almacenamiento sustancial.
- **Polimorfismo y sobrecarga (override) de operadores.** Uso de funciones iguales para diferentes clases.
- **Creación de versiones.** Existencia de varias versiones del mismo objeto.

## 8.2 Características de las bases de datos orientadas a objetos

Las bases de datos orientadas a objetos se diseñan para trabajar en conjunción con lenguajes de programación orientados a objetos como Java, C++ y C# y utilizan exactamente su mismo modelo estructural. Son una buena elección para aquellos sistemas que necesitan un buen rendimiento en la manipulación de tipos de datos complejos.

Los **ODBMS** (sistema gestor de base de datos orientada a objetos) proporcionan los costes de desarrollo más bajos y el mejor rendimiento cuando se utilizan objetos, ya que almacenan objetos en disco y tienen una integración transparente con el programa escrito en un lenguaje de programación orientado a objetos, al almacenar exactamente el modelo de objeto utilizado a nivel aplicativo, lo que reduce los costes de desarrollo y mantenimiento.

## Recuerda

**Cuando se integran las características de una base de datos con las de un lenguaje de programación orientado a objetos, el resultado es un sistema gestor de base de datos orientada a objetos (ODBMS).**

Los ODBMS pueden incorporar diferentes prestaciones, y sus características se pueden englobar en tres grandes bloques:

- **Mandatorias.** Aquellas características del sistema destinadas a tener un sistema de BDOO sólido. Entre otras, podemos destacar el soporte para objetos complejos, asignación de identidad de objetos, capacidad de encapsulación o el soporte de diferentes tipos o clases.
- **Opcionales.** Aquellas que pueden ser añadidas para la mejora del sistema pero que no son obligatorias, como la herencia múltiple o las versiones de objetos.
- **Abiertas.** Definen el tipo de sistema y su uniformidad, y su definición dependerá según el tipo de aplicación al que esté destinada la Base de Datos.

### 8.3 Instalación del gestor de bases de datos

Los sistemas gestores de bases de datos orientados a objetos (Figura 8.1) no difieren en instalación de los gestores de bases de datos relacionales.

La arquitectura general de cliente-servidor también se encuentra en estos sistemas gestores, donde se instala un servicio al que los programas clientes le hacen peticiones de consulta o actualización, mediante el API nativa de la base de datos, o mediante sentencias SQL. En tal situación, en bases de datos orientadas a objetos como db4o, ObjectStore y Oracle, se deben instalar y configurar como un servicio. Sin embargo, para el uso pedagógico, en este apartado se tratará el sistema gestor de bases de datos orientadas a objetos ObjectDB, el cual es posible instalar como una biblioteca en un proyecto Java.

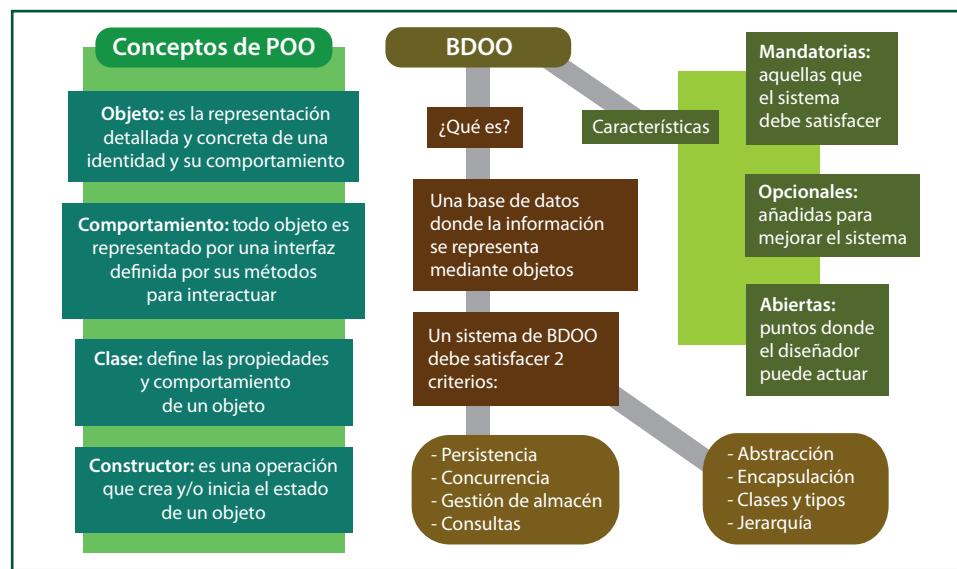


Figura 8.1

Mapa conceptual de bases de datos orientadas a objetos.

Las arquitecturas de configuración posibles son:

- **Modo embebido**

Un único thread que abre el archivo de datos de la OODBMS y que hace las operaciones de E/S.

- **Modo servidor**

Usa el protocolo TCP/IP. Las conexiones se realizan a un servidor de objetos ejecutado en otra máquina virtual. Por cada una de estas conexiones el servidor lanza un thread.

- **Modo servidor embebido**

El servidor y los clientes “viven” dentro de la misma máquina virtual. Especial para aplicaciones que son multihilo pues cada hilo podría abrir una o más conexiones.

También existen posibilidades en cuanto al cómo se almacena la información. Siempre en un único archivo de datos, las arquitecturas pueden ser:

- **Arquitectura de índices B-Tree**

Índices por clase.

Índices por campo (similar a lo que es indexar una tabla en un RDBMS).

- **Marshaller de objetos**

Guarda de manera “inteligente” los objetos padres y sus colecciones dentro del archivo.

Para ello, usa como estrategia un agrupamiento adecuado para mejorar la E/S.

## 8.4 Creación de bases de datos

Un sistema gestor de bases de datos permite gestionar diversas bases de datos.

Un esquema (*schema*, en inglés) se utiliza en las bases de datos como un sistema para describir la estructura, los elementos y las relaciones de la base de datos.

Sin quedarnos con las excepciones, las bases de datos se crean desde el sistema gestor, es decir, el o los programas desde donde las utilizaremos se conectarán a la base de datos, pero ésta ya estará creada.

Es más, normalmente desde programa no se altera el diseño de la base de datos, es decir, de sus tablas, vistas, relaciones ni de cualquier elemento de diseño.

Por lo tanto, será el gestor de base de datos el que nos permita crear y diseñar la base de datos mediante herramientas que nos proporciona el mismo, o bien herramientas implementadas por terceros (Figura 8.2).

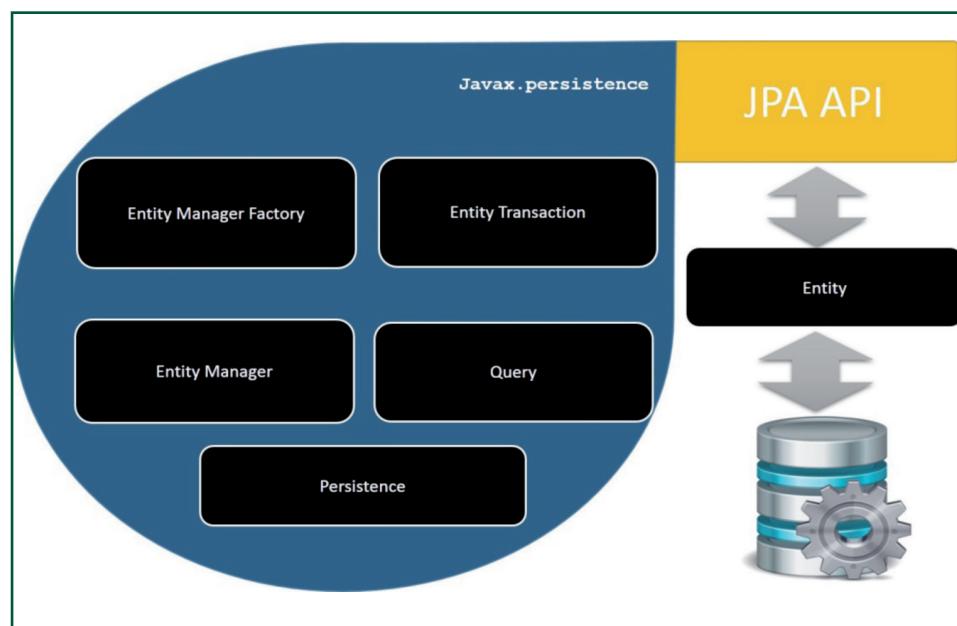
En el caso de las bases de datos orientadas a objetos, estas reglas se mantienen, por lo que se creará la base de datos desde las herramientas del sistema gestor, en nuestro caso ObjectDB. Para poder acceder a los elementos de una base de datos, será necesario utilizar el método `createEntityManager` del paquete `javax.persistence`:

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory ("$objectdb/db/myDb-  
    File.odb");  
EntityManager em = emf.createEntityManager();
```

De forma genérica, la cadena de conexión será: "objectdb://localhost/myDbFile.odb;user=admin;password=admin". De esta forma se podrá acceder posteriormente a los métodos de acceso a los objetos (`remove`, `contains...`) o de consultas (`createQuery`, `createNativeQuery...`).

Para crear una base de datos en blanco, sería:

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("objectdb:myDbFile.  
    tmp;drop");
```



**Figura 8.2**

Clases de arquitectura JPA.

## 8.5 Tipos de datos básicos y estructurados

Los objetos contienen propiedades, que incluyen sus atributos y las relaciones que tienen con otros objetos. Para poder guardar esta información, se requieren tanto datos atómicos como datos estructurados o compuestos (Figura 8.3). Los tipos **atómicos** son los siguientes:

- booleanos: boolean
- caracteres: char, string
- enteros: octect, short, long unsigned short, unsigned lon
- reales: float y double

### Recuerda

Para operar con los datos contenidos en una base de datos se utiliza un lenguaje especial llamado SQL (*Structured Query Language - Lenguaje de Consultas Estructurado*).

Los tipos **estructurados** son los siguientes:

- Date (fecha)
- Time (hora)
- Timestamp (marca de tiempo)
- Interval que representa un período de tiempo

Java type	Database type
String (char, char[])	VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)
Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
int, long, float, double, short, byte	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
byte[]	VARBINARY (BINARY, BLOB)
boolean (Boolean)	BOOLEAN (BIT, SMALLINT, INT, NUMBER)
java.util.Date	TIMESTAMP (DATE, DATETIME)
java.sql.Date	DATE (TIMESTAMP, DATETIME)
java.sql.Time	TIME (TIMESTAMP, DATETIME)
java.sql.Timestamp	TIMESTAMP (DATETIME, DATE)
java.util.Calendar	TIMESTAMP (DATETIME, DATE)
java.lang.Enum	NUMERIC (VARCHAR, CHAR)
java.util.Serializable	VARBINARY (BINARY, BLOB)

**Figura 8.3**  
Mapeo de los tipos de datos con JPA de Java.

## 8.6 El lenguaje de definición de objetos

El **lenguaje de definición de objetos (ODL - Object Definition Language)** es aquel propuesto como estándar para especificar la estructura de las bases de datos en términos orientados a objetos. Se utiliza para la conexión entre

bases de datos y lenguajes de programación como Java, C++ o *Smalltalk*. Se trata de un enfoque estructural que plantea que el desarrollo del sistema de información, llamado **modelo**, se defina de forma completamente independiente de la plataforma en la que se vaya a programar la aplicación. El propósito del **ODL** es permitir que los diseños de bases de datos orientados a objetos sean escritos y puedan ser expresados en declaraciones de un sistema de administración de bases de datos orientado a objetos (**OODBMS - Object Oriented Database Management System**) (Figura 8.4). Las características básicas de ODL son:

- Riqueza expresiva orientada al sistema de información y no al lenguaje de programación.
- Cubre todos los tipos de datos proporcionados por las bases de datos y permite el vínculo entre ellos.
- Ofrece sencillos mecanismos de herencia entre los objetos de la base de datos.

**Figura 8.4**

Ejemplo de creación de dos objetos en lenguaje ODL. Para cada uno de ellos se especifican una serie de características propias.

```
Type Date Tuple {year, day, month}
Type year, day, month integer

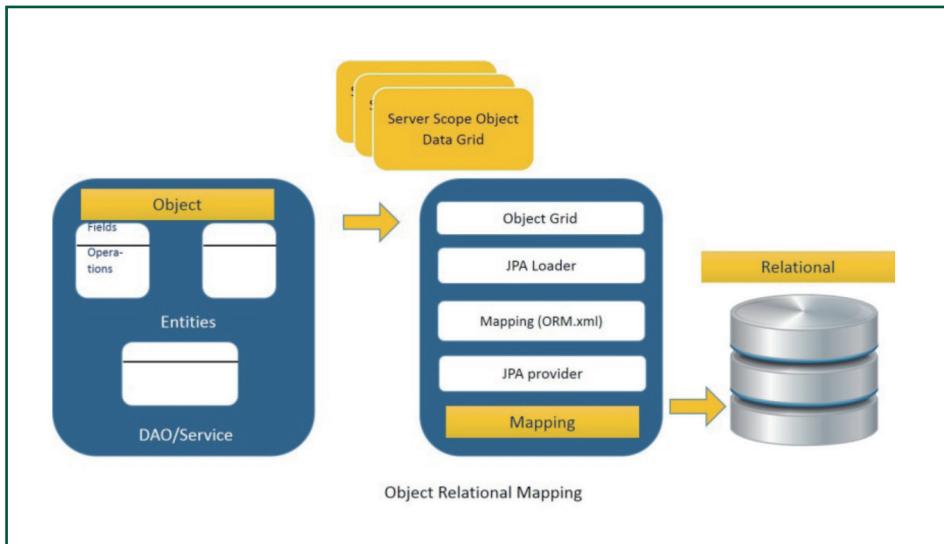
Class Manager
    attributes(id : string unique
              name : string
              phone : string
              set employees : Tuple { [Employee], Start_Date : Date })

Class Employee
    attributes(id : string unique
              name : string
              Start_Date : Date
              manager : [Manager])
```

En nuestro caso, al utilizar Java con la base de datos dbObjects, nos serviremos de JPA (*Java Persistence API*) para crear las entidades y cualquier otro elemento de diseño en la base de datos orientada a objetos (Figura 8.5).

Con esta metodología, integraremos el sistema gestor de base de datos ObjectDB en nuestro proyecto como una biblioteca más, y crearemos clases que “mapearán” las entidades de la base de datos, creándolas en caso de no existir. El mapeo se realiza con anotaciones (@) a nivel de clase y a nivel de propiedad que referenciarán directamente a las tablas y campos de la base de datos.

Veremos en la parte práctica que existen anotaciones para cada uno de los elementos de diseño que podemos tener en una base de datos.

**Figura 8.5**

Arquitectura ORM.

Por ejemplo, la anotación `@Entity` antes de la clase significa que es una tabla, y `@Id` a nivel de propiedad significa que es la clave primaria.

Para poder asociar los objetos con los datos almacenados se utiliza ORM (Object Relational Mapping), que permite pasar los datos tipo objeto a relacional y al revés. Esto se hace mediante un archivo xml y añadiendo, en las clases, propiedades, métodos anotaciones; todas empiezan por el símbolo "@", para indicar el tipo de elemento.

Las más importantes son las siguientes:

Anotación	Descripción
<code>@Entity</code>	Declara la clase como una entidad o una tabla.
<code>@Id</code>	Especifica la propiedad como identidad (es decir, clave principal de una tabla de la clase).
<code>@Column</code>	Especifica que es una columna p.
<code>@JoinColumn</code>	Especifica la entidad asociación o entidad colección. Esto se utiliza en muchos-a-uno y uno-a-muchas asociaciones.
<code>@ManyToMany</code>	Define una relación many-to-many entre tablas.
<code>@ManyToOne</code>	Define una relación de many-to-one entre tablas.
<code>@OneToMany</code>	Define una relación one-to-many entre tablas.
<code>@OneToOne</code>	Define una relación one-to-one entre tablas.
<code>@NamedQueries</code>	Especifica la lista de consultas con nombre.
<code>@NamedQuery</code>	Especifica una consulta con nombre estático.

## Recuerda

**El lenguaje de consultas que proporciona JPA es JPQL, pero este lenguaje es independiente de si la base de datos es relacional u orientada a objetos.**

## 8.7 Mecanismos de consulta

Tal y como ocurre en las bases de datos relacionales, el mecanismo de consulta por excelencia para las bases de datos orientadas a objetos es SQL.

No importa que los datos estén guardados como objetos en vez de tablas relacionales, pues podemos abstraer la estructura de la base de datos orientadas a objetos equiparándola a tablas y relaciones, donde el concepto de tabla vuelve a ser el de Entidad.

En este contexto, Java y en concreto JPA nos proporciona en su lenguaje de anotaciones, expresiones como `@JoinColumn` junto con `@ManyToOne` para especificar una clave foránea y la cardinalidad de la relación.

Con todo ello, una clase con `@Entity` en su inicio es, a la vista de un programa Java, la tabla de la base de datos, y por lo tanto, podemos emplear para consultar sus filas, un lenguaje de consulta que se parecerá mucho al lenguaje SQL que tenemos en las bases de datos relacionales.

Java, y en concreto JPA, proporciona el lenguaje de consultas JPQL (*Java Persistence Query Language - Lenguaje de Consulta de Persistencia en Java*). Este lenguaje tiene las características de SQL y es independiente del sistema gestor de base de datos al cual nos conectemos mediante nuestro programa Java.

Un ejemplo sencillo de consulta JPQL:

```
SELECT p.nombre  
FROM Persona AS p
```

Este ejemplo nos mostraría todas las filas de la entidad (tabla) persona, con una única columna para el nombre.

## 8.8 El lenguaje de consultas: sintaxis, expresiones, operadores

El lenguaje de consultas para el sistema gestor de base de datos empleando Java y JPA corresponde al lenguaje JPQL.

El lenguaje JPQL tiene una sintaxis casi idéntica al lenguaje SQL y, por ello, nos remitimos a la parte práctica de esta unidad para realizar un estudio detallado y, en esta parte, nos limitaremos más a la estructura y generalidades del lenguaje.

Como en SQL, la sentencia de consulta es:

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

Como se mencionó en un apartado anterior, para conectarnos a la base de datos, usaremos un EntityManager:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("$objectdb/db/points2.odb");
EntityManager em = emf.createEntityManager();
```

Para consultar la base de datos, para ejecutar la sentencia SELECT de JPQL, se puede optar por crear una anotación @NamedQuery dentro de la clase de la entidad, justo después de @Entity.

```
@NamedQueries({
    @NamedQuery(name = "Point.dameUno", query = "SELECT p FROM Point p WHERE p.x = 692")})
```

Y, en este caso, la llamada para la ejecución desde el programa principal será:

```
Query q = em.createNamedQuery("Point.dameUno");
```

O bien crear el query en la propia llamada:

```
TypedQuery<Point> query =
    em.createQuery("SELECT p FROM Point p", Point.class);
```

En el apartado siguiente, veremos cómo recuperar los datos que ha devuelto la consulta y en qué estructura nos los devuelve.

## 8.9 Recuperación, modificación y borrado de información

Como se ha visto en el apartado anterior, consultar la base de datos orientada a objetos mediante JPQL se realiza a través de una sentencia SELECT, ya sea llamando a un query creado en la propia clase entidad, ya sea creándolo en el mismo momento de la llamada, desde el programa principal.

## Recuerda

**La iteración se puede realizar con un for simple o bien con el objeto Iterator que posee cualquier lista y los métodos hasNext() y next(), teniendo en cuenta la conversión de tipos con el casting.**

Una vez se ha ejecutado la sentencia:

```
Query q = em.createNamedQuery("Point.dameUno");
```

O:

```
TypedQuery<Point> q =  
    em.createQuery("SELECT p FROM Point p", Point.class);
```

se pueden recuperar los datos en una lista de objetos de la clase de la entidad. En ambos ejemplos, `List<Point> puntos = q.getResultList();` nos devolverá dicha lista de objetos.

JPA proporciona todas las operaciones necesarias como si de una lenguaje DML (*Data Manipulation Language*) se tratara (Figura 8.6); proporciona operaciones de consulta, como hemos visto; y proporciona operaciones para la funcionalidad CRUD (Create, Read, Update, Delete), es decir, a parte de consultar (R Read), aquellas que modificarán el contenido de la base de datos.

Supongamos que nos conectamos mediante EntityManager a una base de datos:

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("$objectdb/db/points2.odb");  
EntityManager em = emf.createEntityManager()
```

```
public Pet createPet(int idNum, String name, PetType  
type) {  
    em.getTransaction().begin();  
    Pet pet = new Pet(idNum, name, type);  
    em.persist(pet);  
    em.getTransaction().commit();  
    return pet;  
}
```

**Figura 8.6**  
Un ejemplo de gestión  
de transacciones con JPA.

En cualquier caso, JPA obliga a incluir la operación en una transacción con la estructura siguiente:

- Se empieza la transacción  
`em.getTransaction().begin();`
- Operaciones CRUD que se necesiten incluir en la transacción bien sea para agregar un objeto, eliminarlo o actualizar alguna propiedad.
- Si se desea que permanezcan los cambios en la base de datos,  
`em.getTransaction().commit();`

- Si no se desea,  
em.getTransaction().rollback();
- Al final, debemos cerrar las conexiones  
em.close();  
emf.close();

En concreto, las operaciones serán:

- Inserción → em.persist(p); donde p es un objeto de la entidad correspondiente (de la tabla)
- Actualización → se actualiza la propiedad o propiedades que se necesiten de la entidad (el commit hará que persista, es decir, que se actualice la base de datos)
- Eliminación → em.remove(p);

Pero, además, estas operaciones pueden realizarse a través de JPQL, pues este lenguaje también es un DML. Tal y como se hace con SELECT, también se puede utilizar INSERT, UPDATE y DELETE para realizar las operaciones CRUD.

## 8.10 Tipos de datos objeto; atributos y métodos

Como en los lenguajes de programación orientada a objetos, en bases de **datos**, los datos **objeto** definen un conjunto de características pertenecientes a un tipo de dato. Existen diferentes tipos de datos objeto:

- **Large Object (LOB).** Destinados a almacenar datos de gran tamaño. Desde hace ya varios años que se almacenan datos de varios Mbytes de peso, como libros, canciones, mapas de alta resolución o vídeos.
- **User-Defined Data Type (UDT).** Se utilizan cuando es necesario combinar distintos tipos de datos en una sola unidad.
- **Colecciones.** También conocidos como *conjuntos*. Este tipo de datos definen una estructura capaz de almacenar elementos que pueden aparecer varias veces.

Como si fueran objetos clásicos de la programación orientada a objetos, a estos tipos de datos objeto se les puede asignar atributos y métodos con el fin de poder otorgarles determinadas características y definir funciones.

## 8.11 Herencia

En bases de datos, la **herencia** representa un intento de adaptación de los diagramas relacionales entre datos al paradigma de la programación orientado a objetos.

## Para saber más

**Existen subdivisiones dentro de los diferentes tipos de datos objeto que permiten especificar con mayor precisión el tipo de dato y el espacio en memoria que hay que destinar para ellos.**

La herencia es un tipo de relación entre una entidad “padre” y una entidad “hijo”. La entidad “hijo” hereda todos los atributos y relaciones de la entidad “padre”, por lo que no necesitan ser representadas dos veces en el diagrama. La relación de herencia se representa mediante un esquema triangular que conecta a las entidades a través de sus vértices. La entidad conectada por el vértice superior del triángulo representa la entidad “padre”. Solamente puede existir una entidad “padre”, es decir, la única herencia permitida es de tipo simple. Las entidades “hijo” se conectan por los vértices de la base del triángulo.

La herencia puede hallarse en dos niveles, en el de los tipos de datos o en el de las tablas:

- **Herencia de tipos.** Supongamos que tenemos un tipo de elemento llamado *Persona* al que asignamos dos características: nombre y dirección. Al crear subdivisiones de este elemento, por ejemplo *Profesor* y *Alumno*, la herencia nos permitirá que éstos hereden las características ya asignadas (nombre y dirección) del elemento padre *Persona*.
- **Herencia de tablas.** El mismo concepto es aplicable para las tablas. En una base de datos podemos crear subtablas, o tablas que derivan de otras. Para el ejemplo anterior, una tabla llamada *Persona*, puede derivar en otras dos tablas dedicadas a *Profesores* y *Alumnos*, y compartir sus características ya definidas.

## 8.12 Constructores

En bases de datos, los constructores nos permiten crear objetos complejos, como los arrays. Un array puede definirse como un grupo o una colección finita, homogénea y ordenada de elementos. Pueden ser de los siguientes tipos:

- **De una dimensión.** Se trata de un tipo de datos estructurado que está formado por una colección finita y ordenada de datos del mismo tipo. Cada uno de los elementos contenidos posee un índice propio, lo que nos permite el acceso a ellos de un modo directo.
- **De dos dimensiones.** Se trata también de un tipo de dato estructurado, finito, ordenado y homogéneo. Su filosofía es la misma que la de un array bidimensional. El acceso a sus elementos es también de modo directo. Se utilizan para representar datos que pueden verse como una tabla con filas y columnas. La primera dimensión representa las columnas, cada elemento contiene un valor y cada dimensión representa una relación entre ellos.
- **Multidimensionales.** Es también un tipo de dato estructurado. Está compuesto por un número determinado de dimensiones mayor a 2. Para hacer referencia a cada componente del array, es necesario utilizar un número “n” de índices, el mismo que el número de dimensiones definidas para él.

Como en la programación, el constructor es el método que inicializa el objeto.

En las BDOO, los objetos suelen ser complejos, por lo que suelen construirse a partir de otros objetos más simples y pueden contener conjuntos de valores almacenados en colecciones.

El constructor atómico es similar al utilizado en POO y permite inicializar el objeto con los valores básicos que puede tener un objeto (como números enteros, reales, booleanos, otros objetos compuestos...).

Sin embargo, existen en ODL otros constructores que permiten definir los tipos de objetos para una base de datos determinada, en los que se pueden añadir tipos complejos basados en colecciones y tuplas, similares a las de los sistemas gestores de bases de datos relacionales, y que, a su vez, se pueden construir a partir de otros más simples, pero, dado su complejidad, no se verán en este tema.

## 8.13 Tipos de datos colección

Para poder almacenar más de un valor en un atributo, como en el caso de que queramos relacionar objetos entre sí, es necesario usar colecciones que permitan almacenar más de un valor. Además también permiten, de forma simple, mantener actualizados los valores mediante actualizaciones, inserciones o eliminaciones.

Las colecciones estándar para las bases de datos orientadas a objetos son:

- **Set<tipo>**. Grupo desordenado de objetos del mismo tipo que no permiten duplicados.
- **Bag<tipo>**. Similar al anterior, pero permite duplicados.
- **List<tipo>**. Grupo ordenado de objetos del mismo tipo que permite duplicados.
- **Array<tipo>**. Grupo ordenado de objetos del mismo tipo al que se pueden acceder por su posición. Es dinámico y los elementos se pueden insertar y borrar en cualquier posición.
- **Dictionary<clave,valor>**. Equivalente a un índice. Utiliza claves ordenadas asociadas a un valor.

Estas colecciones son también muy importantes porque son la forma de realizar las relaciones entre objetos y obtener resultados en consultas.

## Resumen

La gran evolución de los lenguajes de programación orientados a objetos como C++ y Java, más la complejidad de la información que necesitan algunas aplicaciones, como las de diseño asistido por ordenador CAD, las aplicaciones de posicionamiento geográfico, etc., provocaron la evolución de los gestores de bases de datos orientados a objetos.

Al principio, se marcó una serie de características: unas obligatorias (mandatorias), otras opcionales y otras que se dejaron abiertas para que las decidiera el diseñador del sistema.

Estas diferencias, junto con la gran cantidad de gestores de bases de datos existentes del modelo relacional, hacen que, en la actualidad, coexistan los modelos orientados a objetos puros y los modelos "mixtos" (objeto-relacional), que incluyen características de los dos modelos.

El ORM (mapeo objeto-relacional) permite aplicar las técnicas de la programación orientadas a los objetos, pero almacenando la información en un gestor de bases de datos relacional.

JPA (Java Persistence API) es el conjunto de clases que permite a Java acceder a la información de la base de datos.

Los métodos de consulta a la base de datos están soportados por el lenguaje de consultas JPQL, característico de JPA y, en resumen, la sentencia SELECT JPA nos proporciona también las clases y los métodos para realizar las operaciones CRUD (Create, Update, Delete), bien directamente, bien mediante JPQL, con las sentencias equivalentes de SQL.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Los sistemas gestores de bases de datos orientados a objetos difieren en instalación de los gestores de bases de datos relacionales.
2. Un ODBMS hace posible que los objetos de la base de datos aparezcan como objetos de un lenguaje de programación en uno o más lenguajes de programación a los que dé soporte.
3. El modo servidor usa el protocolo TCP/IP. Las conexiones se realizan a un servidor de objetos ejecutado en otra máquina virtual. Por cada una de estas conexiones el servidor lanza un *thread*.
4. Para poder guardar la información, se requieren datos de tipo estructurado o de tipo compuesto.
5. Para operar con los datos contenidos en una base de datos se utiliza un lenguaje especial llamado SQL (*Structured Query Language* - Lenguaje de Consultas Estructurado).
6. El lenguaje de consultas que proporciona JPA es JPQL, pero este lenguaje es independiente de si la base de datos es relacional u orientada a objetos.
7. Para hacer referencia a cada componente del array, es necesario utilizar un número "n" de índices, que es diferente al número de dimensiones definidas para él.

**Completa las siguientes afirmaciones:**

8. Los ODBMS pueden incorporar diferentes prestaciones, y sus características se pueden englobar en tres grandes bloques: \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
9. El lenguaje de definición de objetos (ODL - *Object Definition Language*) es aquel propuesto como \_\_\_\_\_ para especificar la \_\_\_\_\_ de las bases de datos en términos orientados a \_\_\_\_\_.

10. Consultar una base de datos orientada a objetos mediante JPQL se realiza a través de una sentencia \_\_\_\_\_, ya sea llamando a un \_\_\_\_\_ creado en la propia clase entidad, ya sea creándolo en el mismo momento de la llamada desde el programa principal.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## 9. GESTIÓN DE BASES DE DATOS RELACIONALES

En la unidad anterior se han detallado las ventajas de la utilización de las **bases de datos orientadas a objetos**. No obstante, el uso de bases de datos relacionales es más universal por la cantidad de sistemas gestores comerciales y libres, y por la cantidad de aplicaciones desarrolladas que utilizan bases de datos relacionales.

A lo largo de este módulo se ha ido avanzando en las estructuras y herramientas para programar, pero al final cualquier aplicación necesita de la persistencia de los datos, es decir, almacenar los datos de forma estructurada, recuperarlos y utilizarlos, ya sea transformándolos o no.

En esta unidad aprenderemos a manejar el sistema gestor MySQL y cómo conectarlo a nuestras aplicaciones Java en NetBeans. Como se vio en la unidad anterior, es posible utilizar el framework JPA (*Java Persistence API*) de Java, para conectar nuestra aplicación con bases de datos orientadas a objetos. Pero la potencia de JPA es tal que, de forma transparente, de la misma manera conectaremos nuestra base de datos MySQL (relacional) a nuestra aplicación.

No obstante, JPA no es la única manera de hacerlo, así que veremos la conexión mediante JDBC (*Java Database Connectivity*) que permite el diálogo con cualquier base de datos relacional, siempre que hayamos instalado el driver correspondiente en la misma, en nuestro caso en MySQL.

A lo largo de la unidad, aprenderemos a conectarnos a la base de datos relacional, a recuperar datos y transformarlos. También se detallará la manera de insertar, modificar y eliminar registros en las tablas de la base de datos.

Comprenderemos las ventajas de utilizar un asistente para gestionar bases de datos con MySQL, observaremos cómo manipular una base de datos desde PHP y finalmente veremos algunas sentencias de MySQL que nos permitirán analizar las características principales y el estado de las propias bases de datos como tales.

### 9.1 Establecimiento de conexiones

Java proporciona la biblioteca JDBC (*Java Database Connectivity*, conectividad Java a bases de datos) como mecanismo común para el acceso a cualquier tipo de BD, independientemente del fabricante.

Para el desarrollador, la BD real que hay detrás es totalmente transparente y obvia la necesidad de realizar ningún tipo de configuración en la máquina donde se ejecuta la aplicación que accede a los datos. Esta biblioteca se encuentra en el paquete *java.sql*. Partiendo de la suposición de que ya hay un SGBD correctamente configurado en un equipo, los pasos que debe hacer el código de la aplicación para acceder a ellos mediante JDBC son:

- 1) Cargar el controlador (*driver*) del SGBD.
- 2) Establecer la conexión a la BD.
- 3) Ejecutar sentencias SQL en la BD y procesar las respuestas.
- 4) Cuando ya no se quiere trabajar más con la BD, cerrar la conexión.

Sólo a título de introducción, para hacerse una idea del significado de cada paso, a continuación se muestra un fragmento de código para la conexión:

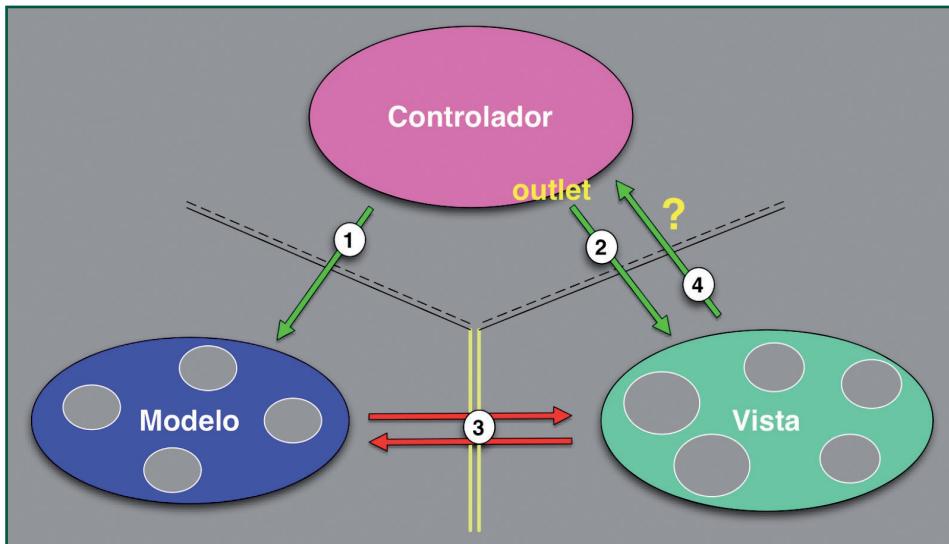
```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(
    "jdbc:myDriver:myDatabase",username,password);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT .... FROM ...");
while (rs.next()) {
    ...
}
con.close();

Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(
    "jdbc:myDriver:myDatabase",username,password);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT .... FROM ...");
while (rs.next()) {
    ...
}
con.close()
```

La conexión se establece con *getconnection* de la clase *DriverManager* (Figura 9.1). Para el caso de MySQL:

```
DriverManager.getConnection("jdbc:mysql://host:puerto/baseDatos","usuario",
    "contraseña");
```

donde el puerto normalmente es 3306 para MySQL.

**Figura 9.1**

En el Modelo Vista Controlador, las clases del Modelo se encargan de la conexión.

## 9.2 Recuperación de información

Una vez nos hemos conectado a la base de datos relacional, se pueden ejecutar consultas SQL con una sintaxis muy similar al estándar.

Todas las interacciones con la BD prácticamente siempre se realizan mediante el envío y ejecución de sentencias SQL y el procesamiento de las respuestas. Las sentencias SQL toman la forma de objetos de la clase Statement, los cuales sólo se pueden instanciar mediante la llamada al método de la clase Connection:

```
Statement createStatement(int tipo, int concurrencia) throws SQLException
```

Los parámetros de entrada especifican cuáles serán las propiedades de las respuestas de la ejecución de la sentencia, los objetos de la clase ResultSet. De hecho, esta clase es la que define el conjunto de constantes estáticas que se pueden pasar como parámetros al crear la sentencia.

El parámetro tipo normalmente suele asignar el valor ResultSet.TYPE\_FORWARD\_ONLY, que indica que la navegabilidad del ResultSet es unidireccional. En el parámetro concurrencia se puede escoger entre dos valores. Si se asigna ResultSet.CONCUR\_READ\_ONLY, el ResultSet resultante será únicamente de lectura, mientras que si se usa el valor ResultSet.CONCUR\_UPDATABLE será de lectura-escritura.

Una vez se ha inicializado la sentencia, el proceso varía según la tarea que se quiere realizar.

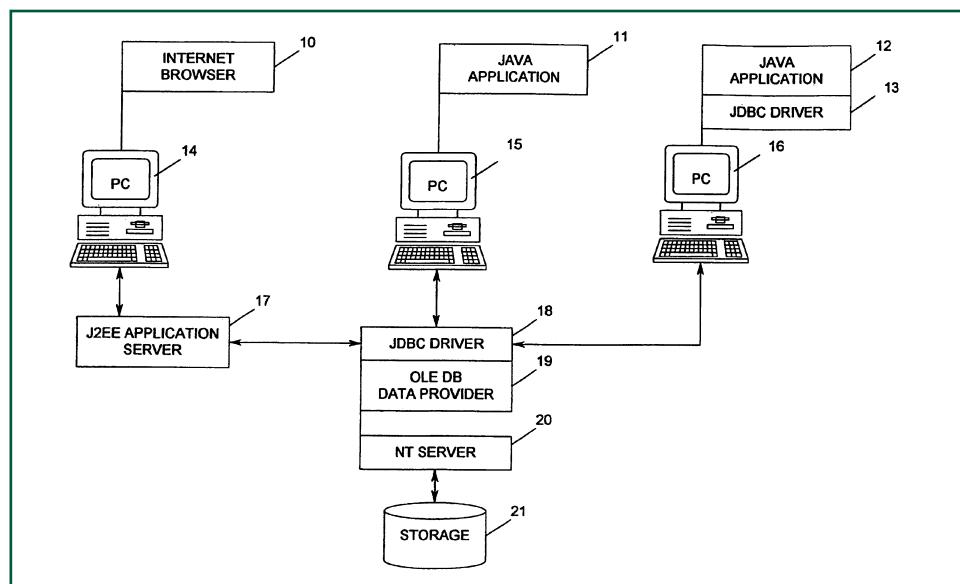
Si se quiere hacer una consulta o modificación sobre unas filas que hay, es necesario llamar sobre la instancia de Statement el método:

```
ResultSet executeQuery(String sql) throws SQLException
```

La cadena de texto sql debe contener una sentencia SELECT válida, que se ejecutará sobre la BD. La instancia de ResultSet devuelta contiene la lista con las filas recuperadas de la BD. Si la consulta no ha obtenido ningún resultado, la instancia de ResultSet estará vacía, nunca se puede devolver null.

ResultSet ofrece los métodos necesarios para navegar por la lista y acceder a los valores almacenados. Mediante el método next () se va posicionando un apuntador interno en cada fila que contiene de manera secuencial, y se devuelve el valor "falso" cuando se ha llegado al final. Una vez posicionados en la fila que se quiere leer, es posible consultar el valor de las celdas para cada columna definida en la tabla. Para ello, hay que llamar al método getXXX adecuado según el tipo de datos de la columna a consultar (Figura 9.2).

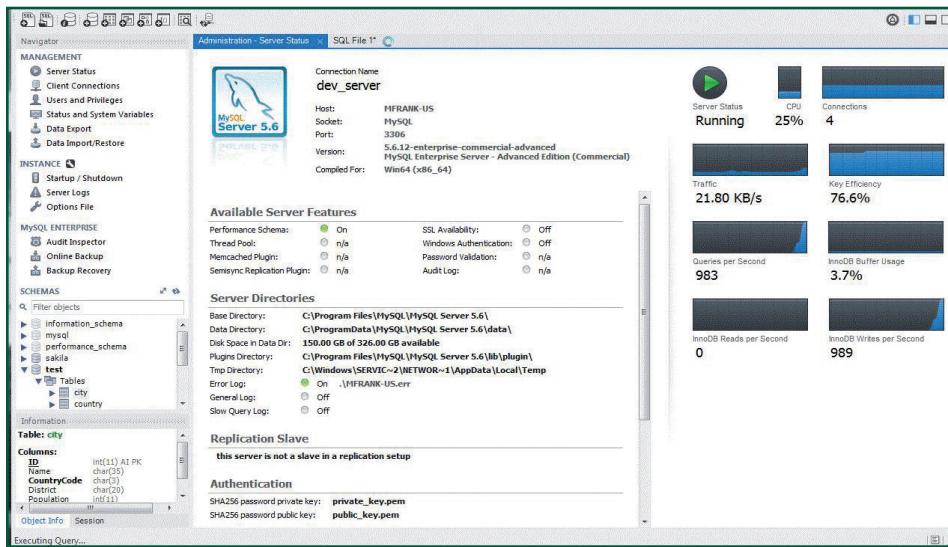
**Figura 9.2**  
Una vez el modelo y el controlador tienen la información, se puede mostrar en diferentes vistas.



### 9.3 Utilización de asistentes

La creación y administración de una base de datos MySQL puede realizarse desde el intérprete de comandos del propio MySQL o programando en Java, pero estos métodos requieren conocimientos avanzados de MySQL, por lo que es fácil cometer errores, que luego serán, además, costosos de detectar.

La solución es utilizar un programa o asistente que facilite esta gestión. El programa más popular es phpMyAdmin, que está escrito en PHP, por lo que se integra perfectamente en cualquier plataforma de desarrollo AMP. También muy popular es MySQL Workbench de Oracle (Figura 9.3), la empresa propietaria de MySQL.



## Recuerda

**Un asistente facilita la creación, administración, explotación y eliminación de bases de datos MySQL.**

**Figura 9.3**

MySQL Workbench.  
Aplicación de MySQL para diseño y administración de bases de datos.

## 9.4 Manipulación de la información

Una vez se ha establecido la conexión con la base de datos, y una vez se ha ejecutado la sentencia SQL, tal y como se ha visto, JDBC nos devolverá al programa los datos en un ResultSet.

A partir de aquí, es responsabilidad del programador utilizar el ResultSet, con sus propiedades y métodos, para el propósito requerido.

Tal y como se ha visto anteriormente, ResultSet ofrece el método next() para posicionararse en la fila siguiente, devolviendo el valor falso cuando llega al final. Para leer una columna concreta de la fila, se utiliza el método getXXX, donde XXX es el tipo de datos de la columna:

tipo getTipo( tipo nombreColumna)

Ejemplo: String getString(String apellido) nos devolverá el apellido de la fila en la que estemos posicionados en el ResultSet.

Una vez posicionados, si se desea modificar el contenido, como se verá más adelante, existen los métodos updateXXX, donde XXX vuelve a ser el tipo de datos:

```
tipo updateTipo(tipo nombreColumna, tipo valorDeActualización)
```

Si el valor almacenado en la columna no se corresponde con el tipo de datos del método llamado, se producirá un error.

En tal situación, una vez se tiene un Statement st, si obtenemos el resultado en un ResultSet:

```
ResultSet rs = st.executeQuery("SELECT ...")
```

parece que la estructura de control óptima para tratar el resultset y obtener las filas es:

```
while(rs.next()){
    ...
}
```

## 9.5 Mecanismos de actualización de la base de datos

La combinación de un lenguaje de programación con una base de datos relacional ofrece libertad absoluta para manipular la información contenida en las tablas.

En Java, mediante JDBC existen herramientas para actualizar los datos de las tablas de las bases de datos relacionales. Una vez conectados podemos actualizar dichos datos. Si a la hora de instanciar al objeto Statement se ha definido que el ResultSet devuelto es de lectura-escritura, entonces también es posible modificar el contenido de las filas que contiene. Como con una simple consulta, hay que avanzar hasta la fila a modificar. En este caso, los métodos de escritura se llaman updateXXX, de manera análoga a los de lectura:

- void updateString (String nombreColumna, String s).
- void update (String nombreColumna, int y).
- void updateDate (String nombreColumna, java.sql.Date d).
- etc.

Una vez efectuadas las actualizaciones de la fila a modificar, se debe llamar al método updateRow() del ResultSet.

Para crear o eliminar elementos dentro de la BD, nuevas filas o una tabla entera, el método a llamar sobre la instancia de Statement es otro:

```
int executeUpdate(String sql) throws SQLException
```

En la cadena de texto de la sentencia, se debe indicar la sentencia SQL que contiene la operación de actualización deseada (Figura 9.4).

Por ejemplo, insertar un nuevo registro:

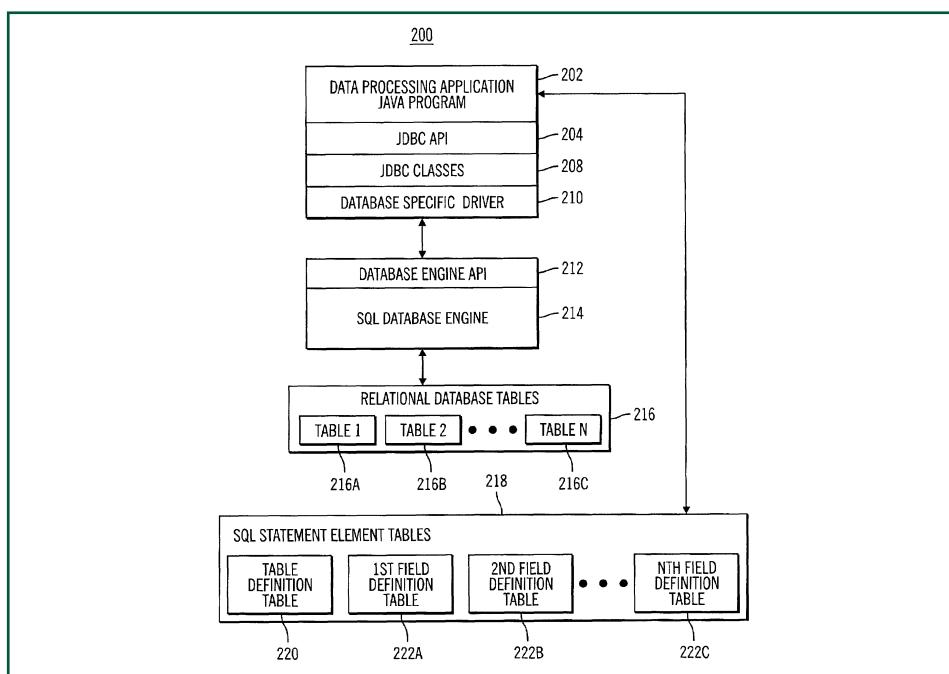
```
INSERT INTO CLIENTS VALUES (1,'El mejor cliente',...)
```

Por ejemplo, para eliminar un registro:

```
DELETE FROM CLIENTS WHERE NOM='El mejor cliente'
```

En este caso no devuelve ningún resultset.

Además, no olvidemos que podríamos ejecutar sentencias que modifiquen el diseño de la base de datos, como CREATE TABLE, ALTER TABLE, DROP TABLE, etc.



**Figura 9.4**

Para que la actualización sea posible, intervienen diversas API.

## 9.6 Ejecución de consultas sobre la base de datos

MySQL admite las siguientes consultas, que devuelven tablas con información relativa a la propia base de datos:

- **SHOW DATABASES.** Muestra las bases de datos que están siendo gestionadas por el servidor MySQL.

- **SELECT DATABASE()**. Muestra la base de datos que actualmente está seleccionada para trabajar sobre ella.
- **SHOW TABLES**. Muestra las tablas que forman la base de datos seleccionada.
- **DESCRIBE Nombre\_de\_tabla**. Describe el diseño de la tabla, es decir, para cada columna muestra el tipo de datos, si admite valores nulos, si forma parte de la clave y, si corresponde, cuál es su valor predeterminado (Figura 9.5).
- **SHOW INDEX FROM Nombre\_de\_tabla**. Si la tabla cuyo nombre se referencia tiene índices, entonces muestra información sobre ellos.

The screenshot shows the MySQL Workbench interface. On the left is the Object Browser with a tree view of schemas and tables. The 'contactos' schema is expanded, showing its tables: contacto, empresa, Views, Routines, ispdcna, joomla, moodle, musica, my\_site\_db, mydb, puntdiet, puntebook, punfies, and puntme. The 'contacto' table is selected. The main area has a 'Query 1' tab with the SQL command: '1 • describe contactos.contacto'. Below the query is a 'Query 1 Result' table showing the table's structure:

Field	Type	Null	Key	Default	Extra
NIF	char(9)	NO	PRI	NULL	
empresa	char(10)	NO	MUL	NULL	
nombre	varchar(45)	NO		NULL	
apellido1	varchar(45)	NO		NULL	
apellido2	varchar(45)	YES		NULL	
telefono	char(20)	NO		NULL	

Below the table, a status message reads: 'Fetched 6 records. Duration: 0.016 sec, fetched in: 0.000 sec'.

**Figura 9.5**

Resultado de una consulta de descripción de tabla (Describe).

## Resumen

El primer proceso de nuestra aplicación para poder trabajar con la base de datos relacional comienza mediante el establecimiento una conexión. En el caso de MySQL desde Java, se efectúa declarando el controlador y ejecutando la conexión, que después deberá cerrarse.

Para la recuperación de la información de la base de datos, se ejecutan sentencias SQL. Primero se preparan con un Statement y después se ejecuta la sentencia SQL que devolverá los datos en un ResulSet.

Los asistentes más comunes para la gestión de bases de datos relacionales como MySQL son PHP-MyAdmin y MySQL WorkBench.

Una vez obtenido un ResulSet, la manipulación de la información se realiza mediante le posicionamiento en el mismo a través del método next() que devuelve un Object que el programador puede transformar (casting) en el tipo de datos adecuado para su aplicación.

La actualización de la información de la base de datos para operaciones de inserción, borrado y modificación se efectúa con executeUpdate, pasando sentencias SQL como INSERT INTO. Pero no se debe olvidar que, con la misma sentencia, se puede alterar el diseño de la base de datos pasándole sentencias como CREATE TABLE.

Los comandos Show Databases, Show Table y Describe se utilizan para conocer el diseño de las propias bases de datos que tenemos instaladas.

## Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. En el parámetro concurrencia se puede escoger entre dos valores. Si se asigna ResultSet.CONCUR\_READ\_ONLY, el ResultSet resultante será de lectura-escritura, mientras que si se usa el valor ResultSet.CONCUR\_UPDATABLE será únicamente de lectura.
2. La combinación de un lenguaje de programación con una base de datos relacional opone restricciones para manipular la información contenida en las tablas.
3. Todas las interacciones con la BD prácticamente siempre se realizan mediante el envío y la ejecución de sentencias SQL y el procesamiento de las respuestas.
4. Los parámetros de entrada especifican cuáles serán las propiedades de las respuestas de la ejecución de la sentencia, los objetos de la clase ResultSet. De hecho, esta clase es la que define el conjunto de constantes estáticas que se pueden pasar como parámetros al crear la sentencia.
5. La creación y administración de una base de datos MySQL no puede realizarse desde el intérprete de comandos del propio MySQL, solo puede hacerse programando en Java.
6. Si se quiere hacer una consulta o modificación sobre unas filas, se deberá llamar sobre la instancia de Statement el método: ResultSet executeQuery(String sql) throws SQLException.
7. Los asistentes más comunes para la gestión de bases de datos relacionales como MySQL son SELECT DATABASE y MySQL WorkBench.

**Completa las siguientes afirmaciones:**

8. Una vez se ha establecido la conexión con la base de datos, y una vez se ha ejecutado la sentencia \_\_\_\_\_, JDBC nos devolverá al programa los datos en un \_\_\_\_\_.
9. La actualización de la información de la base de datos para operaciones de inserción, borrado y \_\_\_\_\_ se efectúa \_\_\_\_\_, pasando sentencias SQL como \_\_\_\_\_.

10. Partiendo de la suposición de que ya hay un SGBD correctamente configurado en un equipo, los pasos que debe hacer el código de la aplicación para acceder a ellos mediante JDBC son: cargar el \_\_\_\_\_ del SGBD, establecer la \_\_\_\_\_ a la BD, ejecutar sentencias \_\_\_\_\_ en la BD y procesar las respuestas.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de este módulo. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

## Soluciones de los ejercicios de autocomprobación

### Unidad 1

1. V
2. F. El entorno de desarrollo integrado (IDE) es un programa informático que ofrece un conjunto de herramientas y recursos necesarios para el desarrollo de aplicaciones informáticas en uno o más lenguajes de programación.
3. V
4. V
5. F. Toda variable deberá estar definida en el bloque de declaraciones de su ámbito, deberá tener un nombre que la identifique de forma única en su ámbito, en la mayoría de los lenguajes se indicará el tipo de dato y, opcionalmente, se le puede asignar un valor inicial.
6. F. Los compiladores informáticos tienen diferentes maneras de interpretar los datos que intervienen en la ejecución de un programa.
7. V
8. programas, hardware, control, organización.
9. sintaxis, programa, lenguaje.
10. Java, C++, bajos, operadores.

### Unidad 2

1. F. En programación, un objeto es una entidad provista de un conjunto de propiedades (datos) y de comportamientos (métodos) que reaccionan según determinados eventos.
2. V
3. V
4. F. Denominamos parámetros a los datos usados por un método para realizar las acciones para las que ha sido definido. La recepción de estos datos le permitirá devolver un valor resultante que podrá ser almacenado en una variable o utilizado en una expresión.
5. V
6. V
7. V
8. clase, métodos, objetos.
9. *Return, variable, area.*
10. instanciando, constructor, clase.

### Unidad 3

1. F. Se pueden incluir varias cláusulas *catch* con distintos tipos de excepciones, aunque siempre se deben poner de mayor a menor detalle.
2. V
3. V
4. F. La estructura de repetición *for* determina un número máximo de repeticiones durante las que se ejecutará una acción definida. Cuando se alcanza el número de repeticiones indicado, el bucle acaba y pasa a la siguiente línea de código.
5. V
6. V
7. F. La prueba y la depuración son procesos que deben hacerse tanto a lo largo de la escritura del programa, para comprobar la funcionalidad de pequeños bloques de código, como al finalizar la aplicación para comprobar su funcionalidad con respecto a los usuarios.
8. *break, continue* y *return*.
9. selección, *if / else*.
10. código, dos.

### Unidad 4

1. V
2. F. Un principio de la POO es la ocultación, que indica que para modificar o acceder a una variable propia de una clase es preciso utilizar un paso intermedio llamado mensaje.
3. V
4. F. Una propiedad estática (*static*) es aquella que es única a nivel de clase, no de objeto.
5. V
6. V
7. V
8. *Getters, valor*.
9. parámetros, *new, clase*.
10. encapsulación, ocultación, visibilidad.

### Unidad 5

1. F. Los flujos de bytes incluyen los métodos y las instrucciones usados para manipular datos binarios, es decir, legibles sólo por un ordenador.
2. V
3. F. En Java, los flujos predefinidos, o estándar, definen procesos a nivel interno que permanecen abiertos a lo largo de la ejecución de un programa.

4. V
5. V
6. F. La serialización es el proceso a través del cual pueden almacenarse objetos directamente en una secuencia de *bytes*.
7. F. Los controladores de eventos se aplican directamente sobre los objetos definidos en nuestro código.
8. ficheros, bytes, caracteres.
9. *FileReader*, apertura, lectura.
10. evento, objeto.

## Unidad 6

1. F. Una estructura es un grupo de elementos heterogéneos relacionados de forma conveniente con el programador y el usuario del programa.
2. V
3. V
4. V
5. F. Indexación base-n (*n*) es un modo de indexación en la que el índice del primer elemento puede ser elegido libremente. En algunos lenguajes de programación se permite que los índices sean negativos.
6. F. La inicialización directa consiste en asignar manualmente un valor a cada uno de los índices de la matriz.
7. V
8. índice, uno.
9. dinámica, matriz, XML.
10. valores, indexado.

## Unidad 7

1. V
2. V
3. F. Una vez definida una clase como subclase, podremos utilizar y reinterpretar las propiedades heredadas, e incluso ocultarlas para que no se pueda acceder desde el exterior.
4. F. El parámetro *override* indica que, si existe una misma función en una subclase y su clase madre, serán las acciones definidas en la subclase las que prevalecerán.
5. V
6. F. En cuanto a las clases, la especificación *abstract* hace referencia a clases que no permiten su instanciación directa en el código, pero sí, en cambio, que otras clases extiendan de ellas.
7. V

8. composición, herencia, composición, herencia.
9. interfaces, clases.
10. corchetes, subclases.

## Unidad 8

1. F. Los sistemas gestores de bases de datos orientados a objetos no difieren en instalación de los gestores de bases de datos relacionales.
2. V
3. V
4. F. Para poder guardar la información, se requieren tanto datos de tipo atómicos como datos de tipos estructurados o compuestos.
5. V
6. V
7. F. Para hacer referencia a cada componente del array, es necesario utilizar un número "n" de índices, el mismo que el número de dimensiones definidas para él.
8. mandatorias, opcionales y abiertas.
9. estándar, estructura, objetos.
10. SELECT, query.

## Unidad 9

1. F. En el parámetro concurrencia se puede escoger entre dos valores. Si se asigna ResultSet.CONCUR\_READ\_ONLY, el ResultSet resultante será únicamente de lectura, mientras que si se usa el valor ResultSet.CONCUR\_UPDATABLE será de lectura-escritura.
2. F. La combinación de un lenguaje de programación con una base de datos relacional ofrece libertad absoluta para manipular la información contenida en las tablas.
3. V
4. V
5. F. La creación y administración de una base de datos MySQL puede realizarse desde el intérprete de comandos del propio MySQL o programando en Java.
6. V
7. F. Los asistentes más comunes para la gestión de bases de datos relacionales como MySQL son PHPMyAdmin y MySQL WorkBench.
8. SQL, ResulSet.
9. modificación, executeUpdate, INSERT INTO.
10. controlador, conexión, SQL.

## Índice

<b>Introducción al módulo</b>	3
<b>Esquema de contenido</b>	4
<b>1. Identificación de los elementos de un programa informático</b>	7
1.1 Estructura y bloques fundamentales .....	7
1.2 Utilización de los entornos integrados de desarrollo .....	8
1.3 Proyectos y soluciones .....	9
1.4 Datos: naturaleza y tipos .....	10
1.4.1 Variables .....	11
1.4.2 Constantes .....	11
1.4.3 Literales .....	12
1.4.4 Tipos de datos .....	13
1.5 Operadores .....	13
1.6 Expresiones .....	16
1.7 Conversiones de tipo .....	16
1.8 Comentarios .....	17
Resumen .....	18
Ejercicios de autocomprobación .....	19
<b>2. Utilización de objetos</b>	21
2.1 Características de los objetos .....	21
2.2 Instanciación de objetos .....	22
2.3 Utilización de métodos y propiedades .....	23
2.4 Programación de la consola: entrada y salida de información .....	25
2.5 Utilización de métodos estáticos .....	25
2.5.1 Propiedades estáticas .....	25
2.5.2 Métodos estáticos .....	26
2.6 Parámetros y valores devueltos .....	26
2.7 Librerías de objetos .....	28
2.8 Constructores .....	29
2.9 Destrucción de objetos y liberación de memoria .....	29
Resumen .....	30
Ejercicios de autocomprobación .....	31
<b>3. Uso de estructuras de control</b>	33
3.1 Estructuras de selección .....	33
3.1.1 Estructuras de selección en Java .....	34
3.2 Estructuras de repetición .....	35
3.2.1 Estructuras de repetición en Java .....	36

3.3 Estructuras de salto .....	39
3.4 Control de excepciones .....	39
3.4.1 Excepciones en Java .....	40
3.5 Prueba y depuración .....	41
3.6 Documentación .....	41
Resumen .....	43
Ejercicios de autocomprobación .....	44
<b>4. Desarrollo de clases</b>	46
4.1 Estructura y miembros de una clase .....	46
4.1.1 Sintaxis de una clase en Java .....	48
4.2 Creación de atributos .....	48
4.2.1 Sintaxis de atributos en Java .....	48
4.3 Creación de métodos .....	49
4.3.1 Sintaxis de métodos en Java .....	50
4.4 Creación de constructores .....	50
4.4.1 Sintaxis de constructores en Java .....	51
4.5 Encapsulación, ocultamiento y visibilidad .....	51
4.5.1 Sintaxis de visibilidad en Java .....	52
4.6 Utilización de clases y objetos .....	53
4.6.1 Sintaxis de utilización de clases y objetos en Java .....	54
4.7 Utilización de clases heredadas .....	54
4.7.1 Sintaxis de herencia de clases .....	56
4.8 Empaquetado de clases .....	56
Resumen .....	58
Ejercicios de autocomprobación .....	59
<b>5. Lectura y escritura de la información</b>	61
5.1 Concepto de flujo .....	62
5.2 Tipos de flujos. Flujos de bytes y de caracteres .....	62
5.3 Flujos predefinidos .....	63
5.4 Clases relativas a flujos .....	63
5.5 Utilización de flujos .....	63
5.6 Entrada desde teclado .....	64
5.7 Salida a pantalla .....	64
5.8 Aplicaciones del almacenamiento de información en ficheros .....	66
5.9 Ficheros de datos. Registros .....	66
5.9.1 Registros del sistema.....	66
5.9.2 Registros de datos propios.....	67
5.10 Apertura y cierre de ficheros. Modos de acceso .....	68
5.11 Escritura y lectura de información en ficheros .....	68
5.12 Almacenamiento de objetos en ficheros. Persistencia. Serialización .....	69

5.13 Utilización de los sistemas de ficheros .....	70
5.14 Creación y eliminación de ficheros y directorios .....	71
5.15 Creación de interfaces gráficos de usuario utilizando asistentes y herramientas del entorno integrado .....	72
5.16 Interfaces .....	73
5.17 Concepto de evento .....	75
5.18 Creación de controladores de eventos .....	76
5.19 Generación de programas en entorno gráfico .....	76
Resumen .....	80
Ejercicios de autocomprobación .....	81
<b>6. Aplicación de las estructuras de almacenamiento</b>	83
6.1 Estructuras .....	83
6.2 Creación de arrays .....	84
6.3 Inicialización .....	86
6.4 Arrays multidimensionales .....	87
6.5 Cadenas de caracteres .....	89
6.6 Listas .....	89
6.7 Colecciones .....	91
Resumen .....	93
Ejercicios de autocomprobación .....	94
<b>7. Utilización avanzada de clases</b>	96
7.1 Composición de clases .....	96
7.2 Herencia .....	98
7.3 Superclases y subclases .....	100
7.4 Clases y métodos abstractos y finales .....	101
7.5 Sobreescritura de métodos .....	102
7.6 Constructores y herencia .....	103
7.7 Acceso a métodos de la superclase .....	104
7.8 Polimorfismo .....	105
Resumen .....	107
Ejercicios de autocomprobación .....	108
<b>8. Mantenimiento de la persistencia de los objetos</b>	110
8.1 Bases de datos orientadas a objetos .....	111
8.2 Características de las bases de datos orientadas a objetos .....	111
8.3 Instalación del gestor de bases de datos .....	112
8.4 Creación de bases de datos .....	113
8.5 Tipos de datos básicos y estructurados .....	115
8.6 El lenguaje de definición de objetos .....	115
8.7 Mecanismos de consulta .....	118
8.8 El lenguaje de consultas: sintaxis, expresiones, operadores .....	118
8.9 Recuperación, modificación y borrado de información .....	119
8.10 Tipos de datos objeto; atributos y métodos .....	121
8.11 Herencia .....	121

8.12 Constructores .....	122
8.13 Tipos de datos colección .....	123
Resumen .....	124
Ejercicios de autocomprobación .....	125
<b>9. Gestión de bases de datos relacionales</b>	<b>127</b>
9.1 Establecimiento de conexiones .....	127
9.2 Recuperación de información .....	129
9.3 Utilización de asistentes .....	130
9.4 Manipulación de la información .....	131
9.5 Mecanismos de actualización de la base de datos .....	132
9.6 Ejecución de consultas sobre la base de datos .....	133
Resumen .....	135
Ejercicios de autocomprobación .....	136
Soluciones a los ejercicios de autocomprobación .....	138

