Introducción a la programación orientada a objetos

Albert Gavarró Rodríguez

PID_00174495

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Muchos de nosotros utilizamos habitualmente los ordenadores y estamos familiarizados con el concepto *programa*. Dentro del ordenador hay muchos de ellos, y nos ofrecen multitud de funcionalidades: escribir documentos, hacer cálculos, e incluso jugar. Pero, ¿qué es un programa?

Aunque no lo creamos, hay programas en todas partes. Basta con que nos fijemos en una lavadora. Seguro que tiene un programa para la ropa delicada, otro para el algodón e incluso uno que sólo centrifuga. ¿Son iguales estos programas y los del ordenador? Pues, a grandes rasgos, sí. Veamos un par de ejemplos más: las notas que tenemos en nuestras agendas y una receta de cocina también son programas. Sorprendente, ¿no? Mucho más sorprendente es el hecho de que nosotros mismos ejecutamos programas de la misma forma que los ejecuta un ordenador.

Un programa no es más que un conjunto de instrucciones que permiten llevar a cabo una tarea. En el caso de la lavadora, el programador nos ofrece distintos programas entre los que podemos elegir. De nuestra elección, es decir, del programa, depende el conjunto de acciones que llevará a cabo la lavadora una vez encendida. Así, la lavadora puede desempeñar distintas tareas (lavado corto, lavado largo, etc.) que están perfectamente definidas por los distintos programas que nos ofrece. Las notas de nuestra agenda y la receta de cocina, al igual que un programa, también son un conjunto de instrucciones que hay que realizar para llevar a cabo las tareas “conservar el empleo” y “cocinar un plato”. En ambos casos, son las indicaciones escritas las que controlan, de forma más o menos precisa, las acciones que llevamos a cabo. Son programas escritos para que nosotros mismos los ejecutemos.

Estructura de un programa

Un programa (sea informático o no) está compuesto por cuatro partes bien diferenciadas:

- **Código.** Es el conjunto de instrucciones en sí. Normalmente, el código está escrito de forma que sea fácil de entender y manipular por una persona.

- **Memoria.** Ofrece un espacio al programa para almacenar datos y recuperarlos más tarde.

- **Entrada.** Es el conjunto de datos que el programa recibe mientras se ejecuta y que condicionan las acciones que éste realiza y, en consecuencia, los

resultados que genera. Normalmente, los datos proceden del usuario (pulsaciones del teclado, movimientos y pulsaciones del ratón, etc.), pero también pueden venir de otros programas. En este último caso tenemos, por ejemplo, los datos enviados por un servidor web a nuestro navegador.

- **Salida.** Es el conjunto de datos generados en forma de resultado durante la ejecución del programa. Estos datos pueden percibirse como acciones desde el punto de vista del usuario. Los resultados pueden ser, pues, variopintos: un número, una hoja de papel impresa, una imagen en la pantalla, etc.

Volvamos al ejemplo de la lavadora: el código son las distintas acciones que ésta lleva a cabo, que dependen, ni más ni menos, del programa elegido y de las entradas (la cantidad de detergente, suavizante y lejía, y la ropa sucia). La salida, en este caso, está clara: la ropa lavada. Que esté más o menos limpia dependerá del tipo de ropa, de cómo estaba de sucia, del programa que hayamos escogido y de la cantidad de producto que hayamos usado, en fin, de las entradas. Por otro lado, la lavadora también usa la memoria para saber en qué punto está dentro del programa seleccionado, el tiempo que le queda para pasar al siguiente estadio de lavado, etc.

Lenguajes y compiladores

Más arriba hemos mencionado que los programas están escritos de forma que resulten fáciles de entender y manipular por una persona. Sin embargo, no siempre fue así. En los albores de la informática, los programas se escribían en forma de agujeros en unas tarjetas especiales. Con estos agujeros se representaban los datos y el código. Pero pasar de la voluntad del programador a los agujeros y viceversa no era tarea fácil. Afortunadamente, años más tarde aparecieron los denominados lenguajes de programación, que utilizaban una sintaxis muy parecida a la de un lenguaje natural para describir las acciones que debía realizar el ordenador. Estos programas eran interpretados posteriormente por otro programa llamado compilador, que se encargaba de traducir todas esas sentencias a algo más liviano para el ordenador. Esta forma de programar, con pocos cambios, es la que ha llegado a nuestros días.

El lenguaje Java

En las próximas páginas daremos las nociones básicas para empezar a programar un ordenador. Si bien estos fundamentos nos permitirán abordar satisfactoriamente casi cualquier lenguaje de programación, este tutorial utiliza el lenguaje Java como herramienta de aprendizaje. Las razones para escoger Java son diversas:

- Es un lenguaje de los llamados “de alto nivel”, lo que significa que está más cerca del lenguaje natural que otros.

- Está ampliamente difundido: en móviles, PDA, páginas web, ordenadores...
- Es multiplataforma, es decir, un programa escrito en Java puede ejecutarse en casi cualquier máquina.
- Es un lenguaje orientado a objetos, que es quizás el paradigma de programación más cercano al razonamiento humano.

Objetivos

Una vez finalizado el módulo, deberíamos ser capaces de:

1. Entender pequeños programas escritos en Java o en cualquier otro lenguaje imperativo.

2. Crear programas simples escritos en Java.

3. Comprender la misión y el funcionamiento de las distintas estructuras de control de flujo, realizar recorridos y búsquedas.

4. Definir funciones aplicando criterios simples de diseño descendente.

5. Crear clases simples, con sus constructores, atributos y funciones.

6. Entender los mecanismos de extensión y herencia.

7. Utilizar las clases del API de Java.

1. Fundamentos de programación

1.1. Declaración y uso de variables

La memoria de un ordenador permite almacenar y posteriormente recuperar gran cantidad de datos. Cuando se programa con un lenguaje de alto nivel como Java, se utilizan nombres para hacer referencia a los datos que hay en la memoria. En el argot del programador, a estos datos con nombre se les llama **variables**. Por definición, no puede haber dos variables con el mismo nombre.

Para garantizar una cierta coherencia en el uso de las variables, éstas, además de tener nombre, deben ser de un **tipo**. El tipo determina qué datos puede almacenar una variable (texto, números, etc.) y qué operaciones podemos realizar con ella. Empezaremos trabajando con los tipos siguientes:

- **int:** valor entero entre –2147483648 y 2147483647. No admite valores decimales. Por ejemplo, 2001 ó 175000000.
- **double:** valor real sin límite de rango y de precisión arbitraria. Admite valores decimales. Por ejemplo, –0,075 ó 3,141592654.
- **String:** texto de longitud arbitraria. Por ejemplo, "La casa de María" u "¡Hola, mundo!".
- **boolean:** cierto o falso.

Para poder trabajar con una variable, primero tenemos que declararla. Declarar una variable consiste en mencionar su nombre y su tipo. En Java se escribe primero su tipo, después su nombre y se termina la línea con un punto y coma. El punto y coma es importante, porque indica el final de la declaración. Por ejemplo:

```
int contador;
```

declara una variable llamada "contador" de tipo *int*. Como ya hemos visto anteriormente, en esta variable podremos almacenar un valor entre –2147483648 y 2147483647.

Las líneas siguientes:

```
boolean esClientePreferente;
double capital, interés;
```

declaran tres variables: "esClientePreferente", de tipo *boolean*, y "capital" e "interés", ambas de tipo *double*. En la primera podremos almacenar *cierto* o *falso*, lo que nos servirá, por ejemplo, para almacenar si se trata de un cliente preferente o no. En las dos últimas, almacenaremos el capital y el interés que estudiamos darle a ese mismo cliente, que puede ser diferente en virtud de que sea cliente preferente o no. Observad que podemos declarar dos o más variables en una misma línea separando sus nombres por medio de la coma.

La línea siguiente:

```
int valores[];
```

declara un vector de variables de tipo *int* llamado "valores". Un vector es un conjunto de variables del mismo tipo. Utilizaremos "valores[0]" para referirnos a la primera variable, "valores[1]" para referirnos a la segunda, y así sucesivamente.

En Java, se puede dar cualquier nombre a una variable. Sin embargo, se deben respetar unas pocas reglas. Los nombres de las variables:

- No pueden comenzar por un carácter numérico. Por ejemplo, "12" o "1Valor" no son nombres válidos.
- No pueden llamarse igual que los elementos del lenguaje. Por ejemplo, no podemos tener una variable que se llame "int" o "double".
- No pueden contener espacios.

Sin embargo, a diferencia de otros lenguajes, Java sí que permite que el nombre de una variable contenga caracteres acentuados.

Otro punto que se debe tener en cuenta es que Java distingue entre mayúsculas y minúsculas. Esto significa que las variables:

```
double capital, Capital;
```

son diferentes en todos los sentidos.

1.2. El operador de asignación

Una vez declarada una variable, ésta tiene un valor indefinido. Para empezar a trabajar con ella, se necesita asignarle un valor. La asignación de valores a

variables se lleva a cabo mediante el llamado operador de asignación: "=" (el signo de igual). Así pues, las líneas siguientes:

```
int contador;
contador = 0;
```

declaran la variable "contador" y le asignan el valor 0.

Como podemos observar, a la izquierda del operador de asignación se emplaza el nombre de la variable, y a la derecha, el valor que se desea asignarle. La asignación (como la declaración de variables) termina en punto y coma.

Veamos más ejemplos de asignaciones:

```
int suma;
suma = 3 + 2;
```

asigna a la variable "suma" el resultado de sumar 3 y 2 (cinco).

```
int contador;
contador = 0;
contador = contador + 1;
```

asigna, en primer lugar, el valor 0 a la variable "contador", y luego la incrementa en una unidad (1). Es importante destacar que cuando asignamos un valor a una variable sobrescribimos su valor anterior. Así pues, al finalizar el código, la variable "contador" valdrá 1.

```
double interés, capital, tasaInterés, tiempo;
capital = 60000;
tasaInterés = 0.045;
tiempo = 6;
interés = capital * tasaInterés * tiempo;
```

asigna a la variable "interés" los intereses generados por el préstamo de un capital de 60.000 euros a una tasa del 4,5% (expresado en tanto por uno) a seis años. Según la fórmula del interés simple, los intereses serán igual al producto de los tres factores. Como puede apreciarse, el producto se representa mediante el carácter asterisco (*). Al final de la ejecución, "interés" valdrá 16.200.

```
String nombre, frase;
nombre = "María";
frase = "La casa de " + nombre;
```

asigna a la variable “frase” el resultado de concatenar el texto “La casa de ” y la variable “nombre”. Al finalizar la ejecución, “frase” será igual a “La casa de María”. Debemos notar que el texto siempre se escribe encerrado entre comillas dobles (“”).

```
boolean esPositivo;
int número;
número = -5;
es_positivo = número > 0;
```

almacena en la variable “esPositivo” el resultado de evaluar si la variable “número” es mayor que 0. Al finalizar el código, “esPositivo” valdrá *false* (falso).

Si bien la flexibilidad que ofrece el operador de asignación es notable, para que una asignación sea válida debe haber una cierta compatibilidad entre el tipo de la variable y el tipo del valor que se le quiere asignar. Por ejemplo, la asignación siguiente no es válida:

```
int peso;
peso = 58.5;
```

pues intenta asignar un valor decimal a una variable entera.

Tampoco es válida la asignación:

```
String resultado;
resultado = 25;
```

pues intenta asignar un valor entero a una variable de tipo texto.

1.3. Expresiones

Como hemos visto en los ejemplos anteriores, a una variable no sólo se le puede asignar un valor literal o el valor de otra variable. Además, se le puede asignar el valor resultante de una operación. Cuando nos encontramos con una combinación de valores literales, variables y operadores, estamos ante lo que se llama una

expresión. Las expresiones, que siempre generan un valor, están sometidas a unas reglas de evaluación que debemos conocer. Si tenemos, por ejemplo:

```
int cálculo;
cálculo = 2 + 3 * 5;
```

la expresión "2 + 3 * 5" se puede interpretar de dos formas: sumar 2 y 3, y el resultado (5) multiplicarlo por 5 (25), o bien multiplicar 3 por 5, y al resultado (15) sumarle 2 (17). Como podemos ver, de la forma de interpretar la expresión depende el resultado.

El lenguaje Java, como muchos otros lenguajes, define unas prioridades o precedencias entre operadores que dictan el orden en el que se realizarán las operaciones. La tabla "Precedencia de los operadores" muestra los operadores más comunes y su precedencia. Cuanta mayor precedencia, antes se evaluará la operación.

Si aplicamos las reglas de precedencia de la tabla al ejemplo anterior, tenemos que la multiplicación precederá a la suma. En consecuencia, el lenguaje resolverá la expresión multiplicando primero 3 por 5, y sumándole 2 al resultado. Así pues, se asignará el valor 17 a la variable "cálculo".

Precedencia	Operadores	Descripción
Más precedencia	–expr. +expr. !	Signo positivo, signo negativo y no.
	* / %	Multiplicación, y cociente y residuo de una división.
	+ –	Suma/concatenación* y resta.
	< > <= >=	"Más pequeño que", "más grande que", "más pequeño o igual que" y "más grande o igual que".
	== !=	"Igual que" y "diferente que".
	&&	Y
Menos precedencia	||	O

*** El significado del operador "+" cambia dependiendo de los operandos. Aplicado a números realiza una suma; aplicado a texto, una concatenación.**

Tabla 1. Precedencia de los operadores

Como hemos visto, la precedencia entre operadores permite al lenguaje determinar, de forma unívoca, el valor de una expresión. Sin embargo, este mecanismo no es suficiente en el caso de tener en una expresión dos o más operadores con la misma precedencia. Cuando esto sucede, el lenguaje resuelve la expresión evaluando dichos operadores de derecha a izquierda. Es decir, se operarán en el mismo sentido en el que se leen. Por ejemplo, el código:

```
int cálculo;
cálculo = 25 % 7 * 3;
```

calculará primero el residuo resultante de dividir 25 entre 7, y después multiplicará dicho residuo por 3. El resultado será 12.

Supongamos ahora que deseamos sumar 2 y 3, y multiplicar el resultado por 5. Ya hemos visto que el código:

```
int cálculo;
cálculo = 2 + 3 * 5;
```

no responde a nuestras necesidades, pues, por precedencia, se evaluará antes la multiplicación que la suma. Para alterar el orden de evaluación de la expresión, encerraremos la suma entre paréntesis:

```
int cálculo;
cálculo = (2 + 3) * 5;
```

Cuando encerramos parte de una expresión entre paréntesis, esta parte se considera una unidad indivisible y se evalúa independientemente antes de ser operada. Así, volviendo al ejemplo, la expresión se resolverá multiplicando el resultado de evaluar la expresión "2 + 3" por 5. El resultado será 25.

Veamos más ejemplos de expresiones. Las líneas siguientes:

```
double precioConDescuento, precio, descuento;
precio = 15.69; // en euros
descuento = 7; // tanto por ciento
precioConDescuento = precio - precio * descuento / 100;
```

calculan el precio rebajado de un producto. En primer lugar, por tener más precedencia, se hace la multiplicación (precio * descuento), después se divide el resultado por cien, y finalmente, se resta el resultado obtenido del valor de la variable "precio". El resultado será 14,5917.

Las líneas siguientes:

```
int número;
boolean estáEntreCeroYDiez;
número = -5;
estáEntreCeroYDiez = número > 0 && número < 10;
```

determinan si un número dado está entre 0 y 10. Para que esto se cumpla, el número debe ser, a la vez, mayor que 0 y menor que 10. En primer lugar, se comprobará si el número es mayor que 0 (número > 0), y después, si es menor que 10 (número < 10), para, finalmente, mediante el operador && (y), ver si ambas condiciones se cumplen. El resultado será igual a *false* (falso).

Aunque el conocimiento y el buen uso de las reglas de precedencia delatan el buen hacer de un programador, a veces, un paréntesis en el lugar adecuado o el uso de expresiones más simples ayudan a despejar dudas y hacer el código mucho más legible. En consecuencia, debemos llegar a un compromiso entre el uso de la precedencia y los paréntesis, en pro de la legibilidad del código.

1.4. Estructura básica de un programa

El lenguaje Java, como la mayoría de lenguajes de programación, requiere una estructura mínima que acoja el programa y sus componentes. En el caso concreto de Java, a esta estructura mínima se le llama **clase**. Aunque las clases se estudiarán en detalle en el capítulo 2, es necesario conocer esta estructura para empezar a escribir programas funcionales. A continuación, se muestra la estructura mínima de una clase ejecutable:

```
public class Programa {
  public static void main(String [] args) {
    // aquí va el código
  }
}
```

Sin querer entrar en demasiados detalles, el código define una clase (*class*) llamada "Programa" con una función "main" que será el que se ejecutará al iniciar el programa. El nombre de la clase ha sido escogido arbitrariamente (es decir, puede cambiarse); no así el nombre de la función, que siempre deberá llamarse "main". Aunque las funciones se estudiarán en detalle en el apartado 1.7, es necesario adelantar que son simples agrupaciones (funcionales) de código. Este programa deberá guardarse en un fichero que se llame como la clase más el sufijo *.java*; en este caso, "Programa.java". El código del programa deberá ir entre las llaves de la función "main".

El programa siguiente:

```
public class Programa {
  public static void main(String [] args) {
    double distancia, velocidad, tiempo;
    velocidad = 120; // en km/h
    tiempo = 2.5; // en horas
    distancia = velocidad * tiempo; // 300 km
    System.out.println(distancia); // escribe 300
  }
}
```

calcula la distancia recorrida por un bólido a partir de su velocidad y el tiempo que ha estado circulando. El texto precedido por doble barra (//) son comentarios. Los comentarios son simples anotaciones que el programador hace en el código del programa para facilitar su comprensión. Los comentarios no se ejecutan. La última línea usa la función "System.out.println" para escribir el resultado en la pantalla.

Como puede apreciarse, todas las variables se han declarado al inicio del código. A diferencia de otros lenguajes, en Java las variables pueden declararse en cualquier parte del código, siempre y cuando la declaración preceda a su uso. Sin embargo, es una práctica habitual declararlas al principio del código para mejorar su legibilidad.

1.5. Estructuras de control de flujo

Supongamos que se desea escribir un programa que calcule y muestre por pantalla el resultado de la suma de los números pares entre 0 y 10. El programa se podría escribir así:

```
public class SumaPares {
  public static void main(String [] args) {
    int suma;
    suma = 2 + 4 + 6 + 8;
    System.out.println(suma); // escribe 20
  }
}
```

El programa, efectivamente, escribirá "20". Sin embargo, si se deseara conocer la suma de los números pares entre 0 y 10.000, escribirlos todos sería inviable o, cuando menos, demasiado costoso.

La solución a este problema nos la dan las estructuras de control de flujo del programa, que nos permiten repetir instrucciones o ejecutar unas u otras dependiendo de una condición.

Para sumar los números pares entre 0 y 10.000, en primer lugar, necesitamos generar de alguna manera todos esos números pares. Una forma fácil de hacerlo es partiendo del primer par, en este caso el 2, e irle sumando 2 una y otra vez:

```
númeroPar = 2;
númeroPar = númeroPar + 2; // númeroPar = 4
númeroPar = númeroPar + 2; // númeroPar = 6
númeroPar = númeroPar + 2; // númeroPar = 8
```

hasta llegar al último par:

```
númeroPar = númeroPar + 2; // númeroPar = 9998
```

en total, 4.998 líneas de código, mientras "númeroPar" sea más pequeño que 10.000. Afortunadamente, esto se puede reducir a tres líneas usando una estructura *while* (mientras):

```
númeroPar = 2;
while (númeroPar < 10000) { // mientras < 10000
  númeroPar = númeroPar + 2; // siguiente número par
}
```

La estructura *while* repite una serie de instrucciones mientras se cumpla una condición dada. Dicha condición se cierra entre paréntesis a la derecha del *while*, mientras que las instrucciones que se han de repetir se cierran entre llaves ({}). En el ejemplo, se repetirá la suma mientras "númeroPar" sea más pequeño que 10.000.

Volviendo a nuestro empeño por conocer el valor de la suma de todos estos números, una vez localizados tan sólo nos queda irlos sumando. Para hacer esto, declararemos la variable "suma", que irá acumulando el resultado. Sólo debemos tener la precaución de ponerla a cero antes de empezar. El programa queda así:

```
public class SumaPares2 {
  public static void main(String[] args) {
    int suma, númeroPar;

    suma = 0;
    númeroPar = 2; // el primer número par
    while (númeroPar < 10000) { // mientras < 10000
      suma += númeroPar;
      númeroPar += 2; // siguiente número par
    }

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

Al final del programa, se mostrará el resultado por pantalla (24.995.000). Hemos de destacar el uso del operador "+=", que asigna a una variable el valor de ella misma más el valor especificado. Es decir:

```
númeroPar += 2;
```

equivale a:

```
númeroPar = númeroPar + 2;
```

Supongamos ahora que queremos crear un programa que sume los múltiplos de un número dado en un intervalo dado. Por ejemplo, los múltiplos de 122 entre 20.000 y 40.000. El programa que suma los múltiplos de dos tiene una limitación: necesita conocer el primer múltiplo del intervalo para poder generar los demás.

Reconocer el primer número par de un intervalo es muy fácil, pero las cosas se complican bastante cuando es cuestión de encontrar el primer múltiplo de 122.

El programa "SumaPares2" construía cada número par sumando dos al inmediatamente anterior. Una vez generado, lo acumulaba, y así hasta haber agotado todos los números pares del intervalo. Sin embargo, ésta no era la única solución: se podría haber optado por recorrer todos los números del intervalo, determinando cuáles son pares y cuáles no. Si el número es par se acumula; si no, no. Esta dicotomía (si se cumple condición, hacer A; si no, hacer B) se puede modelar mediante la estructura *if/else* (si/si no).

Una forma de determinar si un número dado es par o no, es dividiéndolo por dos y analizando el residuo. Si el residuo es cero, el número es par. Teniendo en cuenta que en Java el residuo de una división se obtiene mediante el operador "%" (por cien), esta condición se podría escribir así:

```
if (número % 2 == 0) {
  // código que hay que ejecutar si es par
}
else {
  // código que hay que ejecutar si no es par
}
```

De modo similar a la estructura *while*, la estructura *if* ejecuta una serie de instrucciones si se cumple una determinada condición. Dicha condición se cierra entre paréntesis a la derecha del *if*, mientras que las instrucciones que se han de ejecutar se cierran entre llaves ({}). Opcionalmente, la estructura *if* acepta una extensión *else*, que especifica una serie de instrucciones que se han de ejecutar en caso de que la condición no se cumpla.

Volviendo al problema, si el número es par, debemos acumularlo, mientras que si no lo es, simplemente lo ignoramos.Utilizando la estructura *if*, el código quedaría así:

```
public class SumaPares3 {
  public static void main(String [] args) {
  int número, suma;
  número = 1; // primer número del intervalo
  suma = 0;
  while (número <= 9999) { // último número del intervalo
    if (número % 2 == 0) { // ¿es par?
      suma += número; // sí, entonces lo suma
    }
    número += 1; // siguiente número del intervalo
  }

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

Al igual que "SumaPares2", "SumaPares3" mostrará 24.995.000 por pantalla. Sin embargo, modificar esta última versión para que opere en otro rango o para que sume los múltiplos de un número diferente de 2, es inmediato. Siguiendo con el ejemplo, si se desea sumar los múltiplos de 122 entre 20.000 y 40.000, basta con cambiar sólo tres líneas:

```
public class SumaMultiples {
  public static void main(String [] args) {
  int número, suma;

  número = 20001; // primer número del intervalo
  suma = 0;
  while (número <= 39999) { // último número
    if (número % 122 == 0) { // ¿es múltiplo de 122?
      suma += número; // sí, entonces lo suma
    }
    número += 1; // siguiente número del intervalo
  }

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

1.6. Esquemas de recorrido y de búsqueda

Hemos visto que cuando se trabaja con estructuras del tipo *while*, se repiten una serie de instrucciones mientras se cumpla una determinada condición. Generalmente, se recurre a las repeticiones con uno de estos dos objetivos:

- Recorrer todos los elementos de un conjunto.
- Buscar un elemento de un conjunto.

Este hecho nos lleva a hablar de esquemas de recorrido y de búsqueda. En el ejemplo "SumaPares3", se analizan los números entre 1 y 9.999. Se comprueba cada número para ver si es par o no y, si lo es, se acumula. No se sale del bucle hasta que no se han analizado todos y cada uno de los números del rango. Estamos pues ante un esquema de recorrido.

Los esquemas de recorrido tienen siempre la misma estructura:

```
while (haya_elementos) {
  hacer_acción(elemento_actual);
  siguiente_elemento();
}
```

Por otro lado, podemos tener repeticiones que tengan como objetivo buscar un determinado elemento dentro de un conjunto. Sin apartarnos de los ejemplos de carácter numérico, supongamos que se desea encontrar el primer múltiplo de 13 entre 789 y 800. Está claro que se deben comprobar los números del rango hasta que se encuentre el primer múltiplo de 13. El código siguiente podría resolver el problema:

```
int número;

número = 789;
while (número % 13 != 0) {
  número++;
}
```

Básicamente, empezando por 789, se va incrementando la variable "número" ("número++" equivale a "número = número + 1") mientras "número" no sea múltiplo de 13. Sin embargo, esta aproximación comete un error: supone que encontrará algún múltiplo dentro del rango. Si este múltiplo no existiera, el bucle *while* continuaría analizando números más allá del rango del problema. Por esto, además de comprobar que no se ha encontrado el número que se buscaba, también se debe comprobar que no se esté saliendo del rango de análisis. El código siguiente tiene en cuenta ambas cosas:

```
int número;

número = 789;
while (número <= 800 && número % 13 != 0) {
  número++;
}
```

Como se puede observar, en primer lugar, se comprueba que "número" no sea mayor que 800 (es decir, que sea más pequeño o igual a 800) y, en segundo lugar, que no sea múltiplo de 13. Al haber dos condiciones que gobiernan el bucle (número <= 800 y número % 13 != 0), se puede salir de él por dos motivos:

- por haber agotado todos los números del rango ("número" ha alcanzado el valor 801) o
- por haber encontrado un múltiplo de 13.

Por esta razón, debemos comprobar, al finalizar el bucle, si se ha salido de él por un motivo u otro, pues de ello va a depender el comportamiento del resto del programa:

```
int número;

número = 789;
while (número <= 800 && número % 13 != 0) {
  número++;
}

if (número % 13 != 0) { // se ha encontrado un múltiplo
  System.out.println(número);
}
else { // no se ha encontrado un múltiplo
  System.out.println("No se ha encontrado.");
}
```

Los esquemas de búsqueda presentan la estructura siguiente:

```
while (haya_elementos && ! elemento_encontrado) {
  siguiente_elemento();
}
```

Es importante observar que la condición "elemento_encontrado" siempre irá negada. Esto siempre será así porque las instrucciones del bucle deben repetirse mientras no se haya encontrado el elemento que se busca.

1.7. Definición y uso de funciones

En nuestro afán por encontrar números de todo tipo, supongamos ahora que deseamos encontrar el primer número primo de un intervalo dado. Hay que recordar que un número primo es aquel que sólo se divide de forma entera por él mismo y por 1. El 13 o el 19, por ejemplo, son números primos.

El siguiente esquema de búsqueda podría resolver el problema:

```
encontrado = es_primo el primer_número_del_intervalo;
while (!encontrado && hay_mas_numeros_en_el_intervalo) {
  encontrado = es_primo el siguiente_número;
}

if encontrado {
  // número encontrado
}
else {
  // número no encontrado
}
```

Probar que un número es primo no es una tarea fácil. De hecho, dado un número, llamémosle *N*, la única forma de comprobar que es primo es dividiéndolo entre todos los números primos entre 2 y *N* – 1, ambos incluidos*. Para que sea primo, los residuos de las divisiones deben ser diferentes de 0. Si hay alguna división cuyo residuo sea 0, el número no es primo.

*** Realmente, hay otras formas mucho más optimas de comprobar la primalidad de un número. Véase http://es.wikipedia.org/wiki/Test_de_primalidad.**

Para simplificar el problema, probaremos la primalidad de un número dividiéndolo entre todos los números menores que él, sean primos o no.

Vamos a generar, en primer lugar, un código que compruebe si un número es primo. Concretamente, intentaremos demostrar que no lo es buscando un divisor entero del mismo. Si fallamos en nuestro intento, el número será primo. Como ya hemos visto, se trata de ir dividiendo el número en cuestión entre todos los números menores que él hasta obtener una división cuyo residuo sea 0. Es decir, debemos implementar un esquema de búsqueda del menor divisor entero:

```
int número, divisor;
boolean esPrimo;

número = 19; // número que queremos comprobar

divisor = 2;
while (número % divisor != 0 && divisor < número) {
```

```
  divisor ++;
}
esPrimo = divisor == número; // resultado: true o false
```

Como en todos los recorridos de búsqueda, hay dos condiciones que gobiernan el *while*: la que comprueba que "divisor" no sea divisor entero de "número" (número % divisor != 0) y la que comprueba que no se haya salido de rango (divisor < número). Consecuentemente, al terminar la ejecución del bucle, "número" será primo si la condición que ha "fallado" ha sido la segunda. En este caso, el número será primo si "divisor" es igual a "número".

Ahora que ya disponemos de un código capaz de determinar si un número dado es primo, vamos a construir el programa que busca el primer número primo de un intervalo. Basándonos en el pseudocódigo propuesto anteriormente, el código siguiente busca el primer número primo entre 10.000 y 10.013:

```
public class BuscarPrimo {
  public static void main(String[] args) {
    int inicio, fin; // inicio y fin del rango
    int número, divisor;
    boolean encontrado;

    // inicio y fin del intervalo de búsqueda
    inicio = 10000;
    fin = 10013;

    System.out.println("Buscando el primer número " +
       " primo entre " + inicio + " y " + fin + "...");
    // comprueba si el primer número del intervalo es
    // primo
    número = inicio; // número que hay que comprobar
    divisor = 2;
    while (número % divisor != 0 && divisor < número) {
      divisor ++;
    }
    encontrado = (divisor == número); // resultado

    // mientras no se haya encontrado un número primo y
    // haya más números en el intervalo
    while (! encontrado && número < fin) {
      // comprueba el siguiente número
```

```
      número ++;
      divisor = 2;
      while (número % divisor != 0 && divisor < número) {
        divisor ++;
      }
      encontrado = (divisor == número); // resultado
    }
    if (encontrado) {
      // si ha encontrado un número primo lo muestra por
      // pantalla
      System.out.println(número);
    }
    else {
      System.out.println("No hay ningún número primo " +
          "en el intervalo");
    }
  }
}
```

Básicamente, el código es un esquema de búsqueda del primer número primo, combinado con el código que comprueba si un número es primo visto anteriormente. Como se puede apreciar, éste último, que hemos enmarcado para distinguirlo con facilidad, se repite dos veces.

Repetir bloques de código dos o más veces dentro de un programa tiene serios inconvenientes:

- **Reduce la legibilidad.** Como se puede apreciar en el programa, cuesta discernir entre el código que pertenece a la búsqueda del número primo y el que comprueba si un número lo es. Las variables se mezclan y las instrucciones también.

- **Dificulta la mantenibilidad.** Si se detecta un error en la función que comprueba si un número es primo o se desea mejorarlo, las modificaciones deben propagarse por todos y cada uno de los bloques repetidos. Cuantas más repeticiones, más difícil será introducir cambios.

- **Es propenso a errores.** Ello es consecuencia de la baja legibilidad y mantenibilidad. Una corrección o mejora que no se ha propagado correctamente por todos los fragmentos repetidos puede generar nuevos errores. La corrección de éstos, a su vez, puede generar otros nuevos, y así indefinidamente.

Al final, la repetición de bloques de código conduce a códigos inestables con un coste de mantenimiento más elevado en cada revisión. Afortunadamente, todos estos inconvenientes se pueden salvar mediante el uso de funciones.

Si nos fijamos en el pseudocódigo que hemos concebido inicialmente, vemos que, de forma natural, hemos detectado la funcionalidad "comprobar si un número es primo", que hemos expresado mediande la fórmula "es_primo". Más adelante, cuando hemos implementado dicha funcionalidad, ha aparecido una variable que podríamos considerar la entrada del subprograma: "número", y otra que se podría considerar su resultado: "esPrimo". En la implementación final, el programa principal ha hecho uso de ambas variables para comunicarse con el subprograma, pasando el número a comprobar a través de la variable "número" y obteniendo el resultado mediante la variable "encontrado" (que es el nuevo nombre que, por razones de legibilidad, hemos dado a la variable "esPrimo").

De todo esto, se derivan dos conclusiones importantes:

- La lógica del subprograma es completamente independiente de la del programa principal. Es decir, al programa principal poco le importa cómo se comprueba si un número es primo, ni que variables se utilizan para hacer dicha comprobación.

- Programa y subpgorama se comunican mediante una interfaz basada en variables bien definida.

Llegados a este punto, podemos declarar lo que se llama una *función*. Una función no es más que una agrupación de código que, a partir de uno o más valores de entrada, o, valga la redundancia, en función de ellos, genera un resultado.

Veamos pues la función “esPrimo”:

```
static boolean esPrimo(int número) {
  int divisor;

  divisor = 2;
  while (número % divisor != 0 && divisor < número) {
    divisor ++;
  }

  return divisor == número; // resultado: true o false
}
```

Al margen del uso del modificador *static*, cuyo significado se estudiará en el capítulo 2, una función se declara escribiendo el tipo de resultado que devuelve (en este caso *boolean*), su nombre (“esPrimo”) y, encerrados entre paréntesis, sus parámetros de entrada separados por comas (“número” de tipo *int*). Como se aprecia en el código, los parámetros de entrada se definen de la misma forma que las variables: en primer lugar se escribe su tipo y, luego,

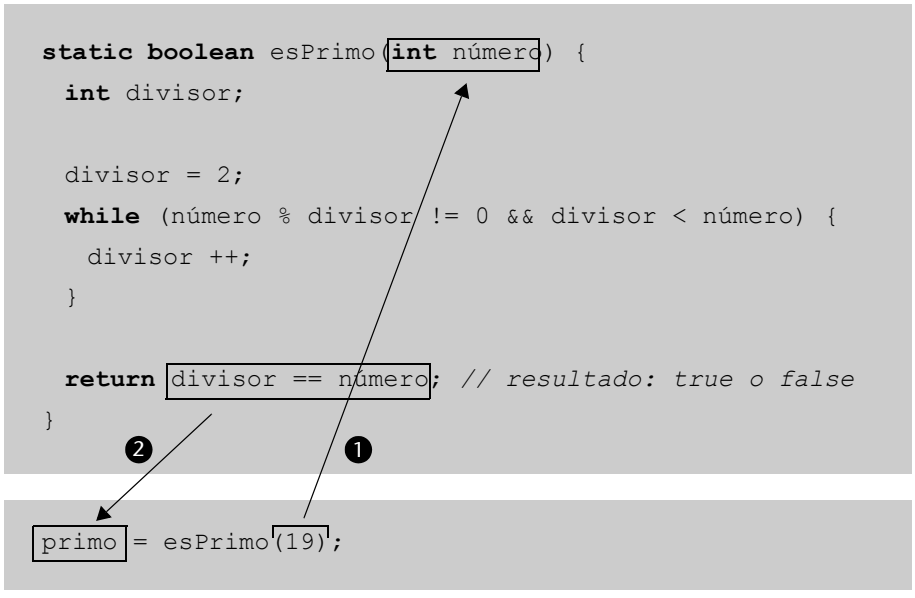
su nombre. El código de la función que sigue a este bloque declarativo va encerrado entre llaves.

Podemos observar que al final de la función aparece una sentencia nueva: *return* (devolver). Cuando se alcanza un *return*, termina la ejecución de la función, y se establece como resultado de la misma el valor de la expresión que sigue al *return*. La sentencia *return* siempre termina en punto y coma.

Una vez definida la función, podemos llamarla desde cualquier parte del código. Por ejemplo, la línea siguiente:

```
primo = esPrimo(19);
```

determina si el número 19 es primo. Dicho de otra manera, asigna a la variable "primo" (de tipo *boolean*) el resultado de la función "esPrimo" para el valor 19. La llamada a una función también termina en punto y coma. Es importante observar que, al ejecutarse la función, el parámetro "número" valdrá 19. De la misma forma, al terminarla, se asignará a la variable "primo" el valor de la expresión que hay a la derecha del *return*:



También es importante destacar que cuando se llama a una función pasa a ejecutarse el código de ésta, y no se vuelve al código que la ha llamado hasta que no se alcanza un *return*.

Si rescribimos el programa "BuscarPrimo" usando funciones, nos queda así:

```
public class BuscarPrimo2 {
  static boolean esPrimo(int número) {
```

```
    int divisor;

    divisor = 2;
    while (número % divisor != 0 && divisor < número) {
      divisor ++;
    }

    return divisor == número; // resultado: true o false
  }

  public static void main(String[] args) {
    int inicio, fin; // inicio y fin del rango
    int número;
    boolean encontrado;

    // inicio y fin del intervalo de búsqueda
    inicio = 10000;
    fin = 10013;

    System.out.println("Buscando el primer número " +
       " primo entre " + inicio + " y " + fin + "...");

    // comprueba si el primer número del intervalo es
    // primo
    número = inicio; // número que hay que comprobar
    encontrado = esPrimo(número); // resultado

    // mientras no se haya encontrado un número primo y
    // haya más números en el intervalo
    while (! es_primo && número <= fin) {
      // comprueba el siguiente número
      número ++;
      encontrado = esPrimo(número); // resultado
    }

    if (encontrado) {
      // si ha encontrado un número primo lo muestra por
      // pantalla
      System.out.println(número");
    }
    else {
      System.out.println("No hay ningún número primo " +
        "en el intervalo");
    }
  }
}
```

Incluso podemos simplificar el recorrido de búsqueda aún más:

```
número = inicio;
while (número <= fin && ! esPrimo(número)) {
  número ++;
}
```

Cuando se trabaja con funciones, deben tenerse en cuenta los aspectos siguientes:

- Los parámetros de una función actúan a todos los efectos como variables. Podemos leerlos y operar con ellos como si fueran variables.

- Los parámetros de una función son una copia de los parámetros de entrada. Si pasamos una variable como parámetro a una función, y dentro del código de la función se modifica el parámetro, el valor de la variable no se verá alterado.

- Al espacio entre llaves que encierra el código de la función se le llama "ámbito de la función". Las variables declaradas dentro de la función sólo pueden verse dentro del ámbito de la función. Es decir, no se puede acceder a ellas desde fuera de la función. Lo mismo es aplicable a los parámetros.

- Una función se caracteriza a partir de su nombre y el tipo de los parámetros de entrada. Si cambia alguno de ellos, estamos ante funciones diferentes. Esto significa que puede haber funciones con el mismo nombre pero con parámetros distintos. A esta característica, que no todos los lenguajes soportan, se le llama "sobrecarga de funciones". A partir del número y el tipo de los parámetros de entrada en una llamada, se averiguará la función a la que se llama.

- Una función puede que no devuelva ningún resultado. En este caso, el tipo de la función será *void* (vacío), y estaremos ante lo que se llama un procedimiento. Un ejemplo de procedimiento és la función *main*, que es la primera función que se ejecuta al iniciar un programa.

2. Programación orientada a objetos

A lo largo de la historia de la informática, han ido apareciendo diferentes paradigmas de programación. En primer lugar, apareció la programación secuencial, que consistía en secuencias de sentencias que se ejecutaban una tras otra. El lenguaje ensamblador o el lenguaje COBOL son lenguajes secuenciales. Entonces no existía el concepto de función, que apareció más adelante en los lenguajes procedimentales, como el BASIC o el C. La evolución no terminó aquí, y continuó hasta el paradigma más extendido en la actualidad: la programación orientada a objetos. Smalltalk, C++ o Java pertenecen a esta nueva generación.

Cada nuevo paradigma ha extendido el anterior, de manera que podemos encontrar las características de un lenguaje secuencial en uno procedimental, y las de uno procedimental en uno orientado a objetos. De hecho, hasta ahora sólo hemos visto la vertiente procedimental del lenguaje Java.

En este capítulo, estudiaremos qué es una clase y qué es un objeto, cómo se definen y cómo se utilizan. Además, conoceremos los aspectos más relevantes de la programación orientada a objetos y las principales diferencias respecto a la programación procedimental.

2.1. Clases y objetos

En la programación orientada a objetos, aparecen por primera vez los conceptos de clase y objeto. Una clase es como una especie de patrón conceptual, mientras que un objeto es la materialización de dicho patrón. Imaginemos la clase "motocicleta". Todos estamos de acuerdo en que todas las motocicletas tienen características comunes que nos permiten distinguirlas como tales. Una motocicleta, entre otras cosas, tiene dos ruedas, carece de techo y tiene un manillar y un motor de una determinada cilindrada. Además, será de alguna marca y tendrá algún nombre de modelo. A este esquema mental, a este patrón, lo llamaremos "clase". Sin embargo, motocicletas hay muchas: la de mi hermano, la del vecino, la del concesionario de enfrente..., todas de diferentes marcas, cilindradas y colores. A esta materialización del patrón le daremos el nombre de "objeto". Clase sólo hay una, pero objetos puede haber muchos.

Una vez aclarada la diferencia entre una clase y un objeto, pasemos a ver cómo declarar una clase. Para nuestro propósito, tomaremos como ejemplo la clase "Producto" para representar cualquier producto que puede venderse en un colmado. Nuestro objetivo, al final, será imprimir un tíquet de compra.

Aunque los productos que podemos encontrar en un colmado tienen características muy dispares, sí que comparten unas pocas que nos permitirán alcan-

zar nuestro objetivo: un código de producto (que corresponderá al código de barras), una descripción y un precio (por ejemplo, un frasco de café soluble, "café de la UOC", con el código 8000534569044 y un precio de 3,50 euros).

Estas características, que llamaremos atributos, se pueden representar como variables. Las líneas siguientes declaran la clase "Producto" y sus atributos:

```
class Producto {
  int código;
  String descripción;
  double precio;
}
```

Como se puede observar, el nombre de la clase va precedido por la palabra reservada *class* (clase). Los atributos se declaran del mismo modo que las variables, aunque encerrados entre llaves.

Cabe recordar que una clase es siempre una simplificación de la realidad. Por esta razón, nunca declararemos atributos para todas y cada una de las características del ente que queramos representar; sólo crearemos aquellos que necesitemos para resolver el problema que se nos plantea.

Por otro lado, podemos tener una o más funciones que lean y escriban esos atributos, por ejemplo, una función que se encargue de leer y otra que se encargue de cambiar (escribir) el precio del producto. Estas funciones, que están estrechamente ligadas a los atributos y que no tienen razón de ser sin ellos, se declaran en el ámbito de la clase.

El código siguiente:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }
}
```

declara la clase "Producto" con dos funciones miembro: "fijarPrecio" y "obtenerPrecio". Es importante observar que ambas funciones tienen visibilidad sobre los atributos de la clase, es decir, pueden acceder a ellos de la misma forma que acceden a sus propios parámetros y variables.

Observad que las funciones que hemos definido carecen del modificador *static* (estático) que hemos visto por primera vez en el apartado 1.7 ("Definición y uso de funciones"). Sin querer entrar en demasiados detalles (cuya discusión va más allá de los objetivos de estos materiales), las funciones que quieran acceder a los atributos de una clase no pueden ser estáticas. En consecuencia, deben prescindir del modificador *static*.

2.2. Instanciación de objetos

Como ya hemos comentado, una clase es sólo un patrón de objetos. Los atributos de una clase no existen en la memoria del ordenador hasta que no se materializan en un objeto. A este proceso de materialización se le llama instanciación.

Un objeto se instancia mediante el operador *new* (nuevo).

El código siguiente:

```
Producto sal;
sal = new Producto();
```

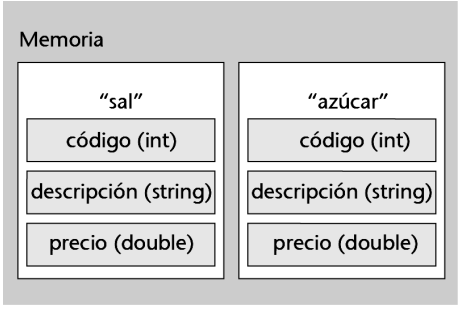
instancia el objeto "sal" de la clase "Producto". Como se puede observar, "sal" debe declararse como una variable del tipo "Producto". Los paréntesis después del nombre de la clase son obligatorios.

A partir de la instanciación, ya tenemos en memoria un objeto con todos sus atributos. Cada nueva instanciación crea un objeto nuevo, independiente de los demás. Si tenemos, por ejemplo, dos objetos, "sal" y "azúcar":

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();
```

en la memoria tendremos dos pares de atributos "código", "descripción" y "precio", uno por cada objeto:



Para acceder a estos atributos, utilizaremos el operador "." (punto).

Las líneas siguientes:

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();

// fija el precio del paquete de sal
sal.precio = 0.60;

// fija el precio del paquete de azúcar
azúcar.precio = 0.81;
```

inicializan el atributo "precio" de los objetos "sal" y "azúcar" a 0,60 y 0,81 respectivamente. Observad la independencia entre ambos objetos. El operador "." nos permite especificar de forma unívoca a qué atributo de qué objeto hacemos referencia.

De la misma forma (usando el operador ".") accederemos a las funciones miembro de la clase. El código siguiente hace exactamente lo mismo que el anterior, pero usando la función "fijarPrecio" que anteriormente habíamos definido:

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();

// fija el precio del paquete de sal
sal.fijarPrecio(0.60);

// fija el precio del paquete de azúcar
azúcar.fijarPrecio(0.81);
```

Agrupándolo todo, el código quedaría así:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  public static void main(String[] args) {
    Producto sal, azúcar;

    sal = new Producto();
    azúcar = new Producto();

    // fija el precio del paquete de sal
    sal.fijarPrecio(0.60);

    // fija el precio del paquete de azúcar
    azúcar.fijarPrecio(0.81);
  }
}
```

Observad que las funciones miembro de una clase, como "fijarPrecio" u "obtenerPrecio", acceden directamente a los atributos de un objeto sin necesidad de especificar a qué objeto pertenecen. Esto es así porque las funciones miembro siempre se ejecutan en el contexto de un objeto. Es decir, cuando nos encontramos con la llamada:

```
sal.fijarPrecio(0.60);
```

ejecutamos la función para el objeto "sal". Entonces, dentro de la función, cada vez que se hace referencia a un atributo, a éste, implícitamente, se le supone de la clase "sal".

2.3. El objeto *this*

Como ya hemos explicado, cuando una función miembro accede a un atributo lo hace en el contexto de un objeto. Por eso no es necesario que especifique a qué objeto hace referencia. Sin embargo, hay casos en los que puede ser interesante disponer de dicho objeto, por ejemplo para pasarlo por parámetro a otra función o para distinguirlo de una variable o un parámetro homónimo.

Las funciones miembro pueden acceder al objeto actual mediante el objeto predefinido *this* (éste). A continuación, se muestra una implementación alternativa (y equivalente) de la función "fijarPrecio":

```
void fijarPrecio(double precio) {
  this.precio = precio;
}
```

En este caso, se utiliza *this* para distinguir entre el atributo de la clase y el parámetro de la función. El "this.precio" hace referencia al atributo "precio" del objeto actual (*this*), mientras que "precio", sin el *this*, hace referencia al parámetro de la función. En este caso, mediante *this* explicitamos a qué objeto hacemos referencia. Entonces, si tenemos la llamada:

```
sal.fijarPrecio(0.60);
```

en el contexto de la función "fijarPrecio", *this* será el mismo objeto que "sal".

2.4. El constructor

Cuando se instancia un objeto, puede ser necesario inicializar sus atributos. Volviendo al ejemplo anterior, establecíamos el precio del producto después de haberlo instanciado, ya fuese accediendo directamente al atributo "precio" o mediante la función "fijarPrecio". Sin embargo, las clases nos ofrecen un mecanismo mucho más elegante de inicializar un objeto: el constructor.

El constructor es una función como cualquier otra, salvo por un par de particularidades: se llama como la clase y no tiene tipo de retorno.

Las líneas siguientes muestran un posible constructor de la clase "Producto":

```
Producto(int código, String descripción, double precio) {
  this.código = código;
  this.descripción = descripción;
  this.precio = precio;
}
```

Observad que el constructor espera tres parámetros: el código, la descripción y el precio del producto. En este caso, la tarea que lleva a cabo el constructor es relativamente sencilla: tan sólo copiar el código, la descripción y el precio proporcionados a los atributos homónimos.

El ejemplo siguiente instancia otra vez los productos "sal" y "azúcar". Sin embargo, esta vez se hace uso del constructor que acabamos de definir:

```
Producto sal, azúcar;

sal = new Producto(80005355, "Sal", 0.60);
azúcar = new Producto(800053588, "Azúcar", 0.81);
```

Agrupándolo todo, el código quedaría así:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // el constructor: inicializa el objeto Producto
  Producto(int código, String descripción, double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  public static void main(String[] args) {
    Producto sal, azúcar;

    sal = new Producto(80005355, "Sal", 0.60);
    azúcar = new Producto(80005388, "Azúcar", 0.81);
```

```
    System.out.println("Precio de 1 paquete de sal: " +
      sal.obtenerPrecio() + " EUR");
    System.out.println("Precio de 1 paquete de azúcar: " +
      azúcar.obtenerPrecio() + " EUR");
  }
}
```

Observad que lo que sigue al operador *new* no es otra cosa que la llamada al constructor de la clase. Después de la instanciación, los atributos de los objetos "sal" y "azúcar" ya estarán inicializados con los valores que hayamos pasado al constructor. El programa generará la salida siguiente:

```
Precio de 1 paquete de sal: 0.60 EUR
Precio de 1 paquete de azúcar: 0.81 EUR
```

Una clase puede tener más de un constructor, del mismo modo que puede tener dos o más funciones con el mismo nombre. Sin embargo, como en el caso de las funciones, los parámetros deben ser distintos.

Si no se define ningún constructor para una clase, ésta tendrá un constructor implícito: el constructor por defecto. El constructor por defecto, que no espera ningún parámetro, es el que usábamos en las primeras implementaciones de la clase "Producto", cuando aún no habíamos definido ningún constructor: 

Es necesario remarcar que el constructor se genera por defecto, en ausencia de constructores explícitos. Esto significa que, si existen constructores explícitos, estamos obligados a usar alguno de ellos para instanciar un objeto. La llamada al constructor por defecto deja de ser válida.

```
azúcar = new Producto(); // llamada al constructor por
                         // defecto
```

2.5. Extensión y herencia

La programación orientada a objetos (POO de ahora en adelante) establece un mecanismo fundamental que nos permite definir una clase en términos de otra: la extensión. La idea subyacente es partir de una clase general para generar una clase más específica que considere alguna característica o funcionalidad no cubierta por la clase más general.

Supongamos la clase "vehículo". Según el diccionario, un vehículo es todo aquello que sirve para transportar personas o cosas de un lugar a otro. Luego, podemos definir los atributos "número de plazas" y "carga máxima" para caracterizar a los vehículos. Sin embargo, hay vehículos con motor y sin motor. Es indudable que los primeros tendrán unos atributos ("poten-

cia", "consumo", etc.) de los que carecerán los segundos y viceversa ("tipo de tracción": animal, humana, etc.). Sin embargo, a la vez, ambos comparten los atributos propios de todo vehículo: "número de plazas" y "carga máxima".

Entonces, la POO nos permite definir las clases "vehículo con motor" y "vehículo sin motor" como clases derivadas (o hijas) de la clase "vehículo". Como clases hijas, heredan los atributos y los comportamientos de la clase madre ("número de plazas" y "carga máxima"), y pueden introducir otros nuevos más específicos ("potencia" y "consumo" en el caso de los vehículos con motor).

Supongamos ahora que en nuestro colmado empezamos a vender productos a granel. La aproximación anterior, en la que cada producto tenía un precio, deja de ser válida para todos los productos. Ahora hay productos que tienen un precio por kilogramo y un peso, además del código y la descripción. Está claro que, en gran medida, los productos que se venden a granel no difieren demasiado de los que se venden por unidades. De hecho, sólo cambia un poco el significado del concepto de precio, que ahora no es por unidad sino por kilo. Además, tenemos un nuevo atributo, el peso, y una nueva forma de computar el precio, el precio (por kilo) multiplicado por el peso.

Jugando con este símil, el código siguiente define la clase "ProductoGranel" como una extensión de la clase "Producto":

```
class ProductoGranel extends Producto {
  // añade el atributo peso (del producto)
  double peso;

  // el constructor también añade el parámetro peso
  ProductoGranel(int código, String descripción, double
    precio, double peso) {
    super(código, descripción, precio);
    this.peso = peso;
  }

  // para los productos vendidos a granel, el precio es
  // igual al resultado de multiplicar el peso (en Kg) por
  // el precio por Kg
  double obtenerPrecio() {
    return precio * peso;
  }
}
```

En primer lugar, observad el uso de la palabra clave *extends* (extiende):

```
class ProductoGranel extends Producto {
```

Mediante esta construcción, definimos una clase en términos de otra. Esto implica que la clase "ProductoGranel" tendrá los mismos atributos y funciones que la clase "Producto", más aquellos que defina de forma expresa. A este mecanismo de transmisión de atributos y funciones se le llama herencia.

Observad también la llamada al constructor de la clase "Producto" desde el constructor de la clase "ProductoGranel" mediante la palabra reservada *super* (superclase):

```
super(código, descripción, precio);
```

Como ya hemos comentado, la clase "ProductoGranel" hereda los atributos y las funciones de la clase "Producto". Esto le permite llamar al constructor de la clase "Producto" para inicializar aquellos atributos que ha heredado. La llamada al constructor de la clase madre, si la hay, debe ser la primera sentencia del constructor de la clase hija.

Finalmente, podemos observar que "ProductoGranel" define una función "obtenerPrecio", que es exactamente la misma que ya existía en la clase "Producto". Sin embargo, su código cambia sustancialmente: ahora el precio se calcula como el producto de los atributos "precio" (por kilo) y "peso":

```
double obtenerPrecio() {
  return precio * peso;
}
```

La nueva función sustituirá a la heredada de "Producto". Este mecanismo nos permite redefinir una función para que se ajuste a las características de la nueva clase. En el caso de la clase "ProductoGranel", este cambio era necesario, pues la función "obtenerPrecio" que había heredado simplemente devolvía el valor del atributo "precio".

El mecanismo de extensión nos permite utilizar objetos de la clase "ProductoGranel" como si fueran de la clase "Producto". El código siguiente imprime el nombre y el precio de varios productos independientemente de su tipo:

```
class Producto {
  int código;
```

```
  String descripción;
  double precio;

  // el constructor: inicializa el objeto Producto
  Producto(int código, String descripción, double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  // devuelve la descripción del producto
  String obtenerDescripción() {
    return descripción;
  }
}

class ProductoGranel extends Producto {
  // añade el atributo peso (del producto)
  double peso;

  // el constructor también añade el parámetro peso
  ProductoGranel(int código, String descripción,
    double precio, double peso) {
    super(código, descripción, precio);
    this.peso = peso;
  }

  // para los productos vendidos a granel, el precio es
  // igual al resultado de multiplicar el peso (en Kg) por
  // el precio por Kg
  double obtenerPrecio() {
    return precio * peso;
  }
}
```

```
public class Caja {
  // muestra por pantalla el precio de un producto
  public static void escribirPrecio(Producto p) {
    System.out.println(p.obtenerDescripción() + " " +
      p.obtenerPrecio() + " EUR");
  }

  public static void main(String[] args) {
    Producto sal;
    ProductoGranel mango, salmón;

    // crea los productos sal, salmón y mango
    sal = new Producto(80005355, "Sal", 0.60);
    salmón = new ProductoGranel(80005373, "Salmón",
      9.55, 0.720);
    mango = new ProductoGranel(80005312, "Mango", 2.99,
      0.820);

    // escribe el precio de los tres productos
    escribirPrecio(sal);
    escribirPrecio((Producto)salmón);
    escribirPrecio((Producto)mango);
  }
}
```

Como podemos observar, en la llamada a la función “escribirPrecio” para los objetos de la clase “ProductoGranel” hay una peculiaridad: el uso del operador de conversión:

```
escribirPrecio((Producto)mango)
```

El operador de conversión permite cambiar el tipo de una variable a otro tipo compatible (*casting*, en inglés). En el ejemplo, forzamos a que “salmón” y “mango” se traten como si fuesen de la clase “Producto”. El operador de conversión toma la forma del tipo destino encerrado entre paréntesis justamente delante de la variable que deseamos convertir.

El resultado será el siguiente:

```
Sal 0.60 EUR
Salmón 6.876 EUR
Mango 2.4518 EUR
```

2.6. Los paquetes y la directiva *import*

Para estructurar el código y evitar conflictos con los nombres de las clases, el lenguaje Java nos permite agrupar las clases en paquetes (*packages*).

Normalmente, los paquetes agrupan clases que están relacionadas por alguna funcionalidad o característica. Por ejemplo, se suelen crear paquetes para agrupar las clases que implementan la interfaz de usuario de una aplicación o aquellas que proporcionan algún tipo de cálculo matemático o probabilístico.

La agrupación de clases en paquetes nos permite:

- Poner de manifiesto la relación entre un conjunto de clases.
- Identificar un conjunto de clases con una funcionalidad.
- Evitar los conflictos con los nombres de las clases: puede haber clases homónimas en paquetes distintos.

Para crear un paquete, basta con escribir en la parte superior de un fichero fuente la palabra *package* seguida del nombre del paquete. Por ejemplo, las líneas siguientes:

```
package postgradosig;
class Asignatura {
  // (...)
```

definen una clase llamada "Asignatura" perteneciente al paquete "postgradosig".

Los nombres de los paquetes tienen las mismas restricciones que los nombres de las variables: no pueden comenzar con un carácter numérico, no pueden llamarse igual que un elemento del lenguaje y no pueden contener espacios.

Los paquetes se pueden dividir en subpaquetes de manera arbitraria, formando una jerarquía de paquetes. Las líneas siguientes:

```
package edu.uoc.postgradosig;
class Asignatura {
  // (...)
}
```

definen una clase llamada "Asignatura" perteneciente al paquete "postgradosig", que se encuentra dentro del paquete "uoc", que a su vez se halla dentro del paquete "edu". Como se puede observar, los paquetes se separan mediante el carácter punto (".").

Cada paquete constituye una unidad léxica independiente dentro del código. Esto significa que los nombres de las clases no traspasan los límites del paquete, lo que permite definir clases homónimas en paquetes distintos. La inde-

pendencia de los paquetes es tal que desde un paquete no se pueden *ver* las clases de otros paquetes, al menos directamente.

Cuando desde una clase se desea acceder a otra clase situada en otro paquete, es necesario indicarlo de forma explícita mediante la directiva *import* (importar). De hecho, esta directiva no se hace otra cosa que importar la definición de la clase.

Supongamos la clase "Alumno" perteneciente al paquete "edu.uoc.postgradosig.alumno" y la clase "Asignatura" perteneciente al paquete "edu.uoc.postgradosig.asignatura". Supongamos también que la clase "Asignatura" tiene una función "matricular" que permite matricular a un alumno a la asignatura. Entonces, el código de la clase "Asignatura" sería parecido al siguiente:

```
package edu.uoc.postgradosig.asignatura;

// importa la clase "Alumno"
import edu.uoc.postgradosig.alumno.Alumno;

class Asignatura {
  // (...)

  public void matricular(Alumno alumno) {
    // (...)
  }
}
```

Observad que a la palabra reservada *import* le sigue el nombre cualificado de la clase, es decir, el nombre del paquete más el nombre de la clase. Es importante remarcar que la directiva *import* no hubiera sido necesaria si las clases hubiesen pertenecido al mismo paquete.

2.7. Visibilidad y encapsulación

Hasta ahora hemos accedido a los atributos y las funciones de un objeto libremente, sin ninguna restricción. Sin embargo, como los atributos contienen el estado de un objeto, no suele ser deseable que código ajeno a la clase pueda acceder a ellos y modificarlos, porque, al desconocer el papel que desempeñan cada uno de los atributos en la implementación de la clase, esto podría dejar el objeto en un estado erróneo.

Se llama encapsulación a la técnica consistente en ocultar el estado de un objeto, es decir, sus atributos, de manera que sólo pueda cambiarse mediante un conjunto de funciones definidas a tal efecto.

Los atributos y las funciones de una clase pueden incorporar en su definición un modificador que especifique la visibilidad de dicho elemento. Hay tres modificadores explícitos en Java:

- ***public***: el elemento es público y puede accederse a él desde cualquier clase,
- ***protected***: el elemento es semiprivado y sólo pueden acceder a él la clase que lo define y las clases que lo extienden, y
- ***private***: el elemento es privado y sólo puede acceder a él la clase que lo define.

Si no se especifica ningún nivel de visibilidad, se aplica la regla de visibilidad implícita (conocida como *package*): el atributo o función será público para todas las clases del mismo paquete, y privado para las demás.

A modo de ejemplo, veamos una nueva versión de la clase "Producto":

```
class Producto {
  private int código;
  private String descripción;
  private double precio;

  // el constructor: inicializa el objeto Producto
  public Producto(int código, String descripción,
    double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  public void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  public double obtenerPrecio() {
    return precio;
  }

  // devuelve la descripción del producto
  public String obtenerDescripción() {
    return descripción;
  }
}
```

Como se puede apreciar, se han declarado todos los atributos privados y las funciones públicas. Esto implica que la sentencia:

```
sal.precio = 0.60;
```

en la que "sal" es una instancia de la clase "Producto", será inválida. En su lugar, debemos usar una llamada a la función "cambiarPrecio" creada a tal efecto:

```
sal.cambiarPrecio(0.60);
```

Es importante remarcar que al menos un constructor de la clase debe ser público. En caso contrario, no se va a poder instanciar ningún objeto de la clase.

3. La API de Java

El lenguaje Java, en su distribución estándar, viene acompañado de una extensa API* (de *Application Programming Interface*, 'interfaz de programación de aplicaciones') que proporciona un conjunto de clases ya implementadas que facilitan o resuelven los problemas más habituales con los que se encuentra un programador.

*** La API puede consultarse en: http://download.oracle.com/javase/1.5.0/docs/api/**

Las clases de la API de Java, que estan estructuradas en paquetes, ofrecen, entre otras, las siguientes funcionalidades:

- lectura y escritura de ficheros en el paquete "java.io",
- representación de gráficos: dibujo de líneas, polígonos, elipses, etc., en el paquete "java.awt",
- creación de interfaces gráficas de usuario en el paquete "javax.swing", y
- gestión y ordenación de objetos mediante pilas, colas, listas, etc., en el paquete "java.util" .

Todas estas clases se pueden importar en nuestro código mediante la directiva *import*. Por ejemplo, el código siguiente hace uso de la clase "Vector" para implementar una cola de tareas que se deben realizar:

```
import java.util.Vector; // importamos la clase Vector

class Tarea {
  private String descripción; // descripción de la tarea

  // constructor de la clase Tarea: su único parámetro es
  // la descripción (textual) de la tarea
  public Tarea(String desc) {
    descripción = desc;
  }

  // devuelve la descripción de la tarea
  public String obtDescripción() {
    return descripción;
  }

  public static void main(String[] args) {
    // el Vector "tareas" almacenará las tareas en orden
    Vector tareas;
    Tarea tarea;
```

```
    // crea una cola de tareas
    tareas = new Vector();

    // agrega tareas a la cola
    tareas.add(new Tarea("Barrer"));
    tareas.add(new Tarea("Fregar"));
    tareas.add(new Tarea("Hacer la compra"));

    // recupera las tareas el mismo orden que fueron
    // agregadas

    // mientras la cola no esté vacía...
    while (! tareas.isEmpty()) {
      // saca el primer elemento (el 0) de la cola...
      tarea = (Tarea)tareas.remove(0);
      // ... y lo muestra por pantalla
      System.out.println(tarea.obtDescripción());
    }
  }
}
```

En el web de la API de Java se puede encontrar la documentación completa de la clase "Vector"*. Entre otras cosas, se explica el funcionamiento de las funciones "remove" e "isEmpty".

*** http://download.oracle.com/javase/1.5.0/docs/api/java/util/Vector.html**

Resumen

En este módulo nos hemos introducido en la programación orientada a objetos de la mano del lenguaje Java. En primer lugar, hemos estudiado los mecanismos comunes a todos los lenguajes imperativos y procedimentales (asignación de variables, estructuras de control de flujo y llamadas a funciones) para continuar con los propios de los lenguajes de orientación a objetos (definición de clase y objeto, extensión y herencia).

Además, se ha detallado la razón y el funcionamiento de las estructuras de control de flujo primordiales, y hemos aprendido a realizar recorridos y búsquedas. Todo esto, combinado con el uso de funciones, nos permitirá estructurar correctamente un código.

Para finalizar, aunque este módulo se ha centrado exclusivamente en el lenguaje Java, las técnicas aquí aprendidas nos permitirán introducirnos con relativa facilidad a otros lenguajes de programación parecidos, como Ada, C, C++, etc.

Bibliografía

Oracle, "*The Java(tm) Tutorials*", http://download.oracle.com/javase/tutorial/.

Oracle, "*Java(tm) 2 Platform Standard Edition 5.0 API Specification*", http://download.oracle.com/javase/1.5.0/docs/api/.