CAPÍTULO 1

Java básico

El objetivo principal de este libro es estudiar técnicas de resolución de problemas que permitan la construcción de programas sofisticados y eficientes. Prácticamente todo el material utilizado se puede aplicar en cualquier lenguaje de programación. Se podría argumentar que para enseñar los conceptos basta con una descripción suficientemente amplia en pseudocódigo de estas técnicas. Sin embargo, creemos que es de vital importancia trabajar con código real.

No hay escasez de lenguajes de programación disponibles. En el momento en que se escribió este libro, C++ era el lenguaje más usado tanto académica como comercialmente. Sin embargo, en 1996, Java irrumpió en la escena como un contendiente viable.

La promoción publicitaria masiva sobre Java se debe en gran medida a su uso como lenguaje en el que se escriben applets. Los *applets* son programas Java ejecutados por un navegador de la World Wide Web, como *Netscape Navigator*. Actualmente, la mayoría de las páginas Web contienen animaciones, muchas de las cuales son applets de Java. Aunque los applets son fantásticos, ellos no son la razón por la que nosotros usamos Java. De hecho, en este libro solamente se estudian los applets en un breve apéndice.

Java proporciona también soporte para la programación concurrente, donde varios procesos se ejecutan en paralelo comunicándose entre sí de forma primitiva. Esta característica es importante en programación más avanzada, pero no se usa en este libro.

El principal atractivo de Java es que se trata de un lenguaje seguro y portable que soporta las construcciones de la programación moderna orientada a objetos. Muchas construcciones de C++ que confunden a los principiantes no existen en Java. Comparado con C++, Java detecta muchos de los errores comunes de programación, ya sea en tiempo de compilación o de ejecución. Java tiene un mecanismo de excepciones que obliga al programador a tratar de forma explícita los errores y un modelo relativamente simple que distingue entre tipos primitivos (como `int`) y tipos definidos por el usuario. Java carece de un tipo puntero explícito.

Java es portable: por ejemplo, un entero tiene el mismo rango de valores en todas las implementaciones de Java, independientemente de la arquitectura subyacente de la computadora. Java proporciona también una herramienta de interfaz gráfica de usuario (GUI) que permite que la entrada y la salida se lleven a cabo a

través de formularios. Aunque no estudiamos esta herramienta en el texto principal (*véase* Apéndice D), es bastante sencilla de utilizar. Lo más importante es que también es portable a cualquier implementación de Java. La filosofía de Java es «escribe una vez, ejecuta en todas partes».

En los primeros cuatro capítulos, estudiaremos las características de Java que se usan a lo largo del libro. Por el contrario, las características y aspectos técnicos que no se van a utilizar, no se cubren.

Quienes deseen una mayor información sobre Java, la encontrarán en los múltiples libros de Java disponibles.

Comenzamos discutiendo la parte del lenguaje semejante a la de un lenguaje de programación de los setenta como Pascal o C. Ésta incluye los tipos primitivos, operaciones básicas, construcciones condicionales y de repetición, y el equivalente en Java de las funciones.

En este capítulo, estudiaremos:

- Algunos conceptos básicos de Java, incluyendo los elementos léxicos simples.
- Los tipos primitivos de Java, incluyendo algunas de las operaciones que pueden llevarse a cabo con las variables de tipo primitivo.
- Cómo se implementan en Java las instrucciones condicionales y los bucles.
- El método estático; el equivalente en Java a la función y el procedimiento.

1.1 El entorno general

¿Cómo se escriben, se compilan y ejecutan los programas Java? La respuesta, por supuesto, depende de la plataforma particular sobre la que se encuentra instalado el compilador de Java. En el Apéndice A se describen algunos de los sistemas de desarrollo más populares.

javac compila los ficheros `.java` y genera ficheros `.class` que contienen *código-j*. *java* invoca al intérprete de Java.

El código fuente de Java se guarda en ficheros con nombres acabados en el sufijo `.java`. El compilador local, *javac*, compila el programa y genera ficheros `.class`, que contienen *código-j*. El código-j es un lenguaje portable intermedio que es interpretado por el intérprete de Java, *java*.

Para los programas de Java, la entrada puede provenir de muchas fuentes:

- El teclado, cuya entrada se denomina *entrada estándar*.
- Parámetros adicionales al invocar el programa ejecutable; *argumentos de la línea de comandos*.
- Una componente GUI.
- Un fichero.

En la Sección 2.4.4 se estudia los argumentos de la línea de comandos que son particularmente importantes para especificar opciones de programa. Java proporciona mecanismos para leer y escribir ficheros. Éstos se estudian en la Sección 2.6.3. Muchos sistemas operativos proporcionan una forma alternativa conocida como *redireccionamiento de fichero*, mediante la cual, el sistema operativo toma la entrada de (o envía la salida a) un fichero, de forma transparente al programa en ejecución. En Unix, por ejemplo, el comando

```
java Programa < ficheroentrada > ficherosalida
```

dispone automáticamente las cosas para que cualquier lectura del teclado sea redirigida para obtenerla de `ficheroentrada` y cualquier escritura en la pantalla sea redirigida sobre `ficherosalida`.

1.2 El primer programa

Comencemos examinando el sencillo programa Java de la Figura 1.1. Este programa imprime una frase corta en la pantalla. Observe que los números de línea mostrados a la izquierda del código *no son parte del programa*. Se proporcionan solamente para facilitar las referencias.

Guarde el programa en el fichero `PrimerPrograma.java` y después compílelo y ejecútelo. En el Apéndice A se describe cómo hacer esto en varias plataformas conocidas. Nótese que el nombre del fichero fuente debe coincidir con el nombre de la clase (mostrado en la línea 4).

1.2.1 Comentarios

Java admite tres formas de comentarios. La primera, heredada de C, comienza con el símbolo `/*` y termina con `*/`. He aquí un ejemplo:

```
/* Este es un comentario
   que ocupa dos líneas */
```

Los comentarios no pueden anidarse.

La segunda forma, heredada de C++, comienza con `//`. No hay símbolo de finalización, sino que el comentario se extiende hasta el final de la línea. Esta forma se ilustra en las líneas 1 y 2 de la Figura 1.1.

Los comentarios hacen el código más legible. Java admite tres tipos de comentarios.

La tercera forma comienza con `/**` en lugar de `/*`. Esta forma se puede usar para proporcionar información a la utilidad *javadoc*, la cual generará documentación a partir de sus comentarios. Esta forma se estudia en la Sección 3.3.

Los comentarios aparecen para hacer el código más legible a las personas. Dichas personas pueden ser otros programadores que tengan que modificar o usar el código que se haya escrito. Un programa bien comentado es signo de un buen programador.

```
1  // Primer programa
2  // MW, 9/1/97
3
4  public class PrimerPrograma
5  {
6     public static void main( String [] args )
7     {
8        System.out.println( "¿Hay alguien ahí fuera?" );
9     }
10 }
```

Figura 1.1 Un primer programa simple.

1.2.2 `main`

Cuando se ejecuta el programa, se llama a la función especial `main`.

Un programa Java está formado por una colección de clases que interactúan entre sí, las cuales contienen métodos. El equivalente en Java a una función o procedimiento es el *método estático*, que se describe en la Sección 1.6. Cuando se ejecuta un programa, se llama al método estático especial `main`. En la línea 6 de la Figura 1.1 se observa que se puede llamar al método estático `main` con argumentos en la línea de comandos. Es obligatorio que los tipos de los parámetros de `main` y el tipo del resultado (`void`) sean los mostrados en la figura.

1.2.3 Salida por pantalla

`println` se utiliza para producir una salida.

El programa de la Figura 1.1 consiste en una única instrucción, mostrada en la línea 8. `println` es el principal mecanismo de salida en Java. En dicha línea se coloca una cadena en la salida estándar `System.out` aplicando un método `println`. La entrada y la salida se estudian con más detalle en la Sección 2.6. Por ahora mencionamos solamente que se usa la misma sintaxis para producir la salida de cualquier entidad, ya sea un entero, un número en coma flotante, una cadena o un valor de cualquier otro tipo.

1.3 Tipos primitivos

Java tiene definidos ocho *tipos primitivos*. Además ofrece al programador una gran flexibilidad para definir nuevos tipos de objetos, llamados *clases*. Sin embargo, los tipos primitivos y los tipos definidos por el usuario poseen diferencias importantes en Java. En esta sección examinaremos los tipos primitivos y las operaciones básicas que se pueden realizar sobre ellos.

1.3.1 Los tipos primitivos

Los tipos primitivos de Java son los enteros, los números en coma flotante, los booleanos y los caracteres.

El estándar Unicode contiene más de 30.000 caracteres codificados, que cubren los principales lenguajes escritos.

Java tiene ocho tipos primitivos, mostrados en la Figura 1.2. El más común es el entero, el cual se especifica mediante la palabra clave `int`. A diferencia de otros lenguajes, el rango de los enteros es independiente de la máquina. Es el mismo en cualquier implementación de Java, independientemente de la arquitectura subyacente. Java también permite entidades de tipo `byte`, `short` y `long`. Los números en coma flotante vienen representados por los tipos `float` y `double`. El tipo `double` tiene más dígitos significativos, por lo que se recomienda su uso en lugar de `float`. El tipo `char` se usa para representar caracteres. Un `char` utiliza 16 bits para representar el estándar Unicode. El estándar Unicode contiene más de 30.000 caracteres codificados, que cubren los principales lenguajes escritos. A bajo nivel, Unicode es idéntico a ASCII. El último tipo primitivo es `boolean`, cuyos valores son `true` y `false`.

Tipo primitivo	Lo que almacena	Rango
`byte`	entero de 8 bits	de -128 a 127
`short`	entero de 16 bits	de -32.768 a 32.767
`int`	entero de 32 bits	de -2.147.483.648 a 2.147.483.647
`long`	entero de 64 bits	de -2^{63} a $2^{63}-1$
`float`	número en coma flotante de 32 bits	6 dígitos significativos (10^{-46}, 10^{38})
`double`	número en coma flotante de 64 bits	15 dígitos significativos (10^{-324}, 10^{308})
`char`	Carácter Unicode	
`boolean`	Variable booleana	`false` y `true`

Figura 1.2 Los ocho tipos primitivos de Java.

1.3.2 Las constantes

Las *constantes enteras* se pueden representar en notación decimal, octal o hexadecimal. La notación octal se indica mediante un `0` delante del número; la notación hexadecimal se indica mediante `0x` o bien `0X` delante del número. Se muestran a continuación las distintas formas posibles de representar el entero 37: `37`, `045`, `0x25`. En este libro no usaremos la notación octal ni la hexadecimal. Sin embargo, debemos tenerlas en cuenta, de forma que usemos solamente un `0` delante de un número cuando pretendamos utilizarlas.

Las constantes enteras se pueden representar en notación *decimal*, *octal* o *hexadecimal*.

Una *constante de tipo carácter* se encierra entre dos comillas simples, como en `'a'`. Internamente este carácter se interpreta como un número pequeño. Las rutinas de salida interpretan posteriormente este número como el carácter correspondiente. *Una constante de tipo cadena* consiste en una secuencia de caracteres encerrados entre dobles comillas, como en `"Hola"`. Hay algunas secuencias especiales, conocidas como *secuencias de escape*, que se usan para representar ciertas constantes de tipo carácter (por ejemplo, ¿cómo se representa un salto de línea de interrogación?). En este libro usamos `'\n'`, `'\\'`, `'\''` y `'\"'`, que representan respectivamente, el carácter de nueva línea, el carácter de barra inclinada hacia atrás, la comilla simple y la comilla doble.

Una *constante de tipo cadena* consiste en una secuencia de caracteres encerrada entre dobles comillas.

Las *secuencias de escape* se usan para representar ciertas constantes de tipo carácter.

1.3.3 Declaración e inicialización de tipos primitivos

Cualquier variable, incluyendo las de tipo primitivo, se declara proporcionando su nombre, su tipo, y, opcionalmente, su valor inicial. El nombre debe ser un *identificador*. Un identificador está formado por cualquier combinación de letras, dígitos y el carácter de subrayado, si bien, no puede empezar con un dígito.

A una variable se le da nombre usando un *identificador*.

Las palabras reservadas, como `int`, no están permitidas. Ni tampoco se deben reutilizar nombres de identificadores que ya están visibles (por ejemplo, no usar `main` como nombre de una entidad).

Java es sensible a las mayúsculas.

Java es *sensible a las mayúsculas*, lo que quiere decir que `Edad` y `edad` son identificadores distintos. Este libro usa la siguiente convención para dar nombre a las variables: todos los identificadores comienzan con una letra minúscula y las palabras nuevas dentro de un identificador, con una letra mayúscula. Un ejemplo es el identificador `salarioMinimo`. A continuación se muestran algunos ejemplos de declaraciones:

```
int num3;                          // Inicialización por defecto
double salarioMinimo = 4.50;       // Inicialización estándar
int x = 0, num1 = 0;               // Declaración de dos variables
int num2 = num1;
```

Idealmente una variable debería estar declarada cerca de su primer uso. Como se mostrará más adelante, el lugar de una declaración determina su ámbito y significado.

1.3.4 Entrada y salida por terminal

La E/S básica por terminal con formato se lleva a cabo mediante `readLine` y `println`. La entrada estándar es `System.in`, y la salida estándar es `System.out`.

El mecanismo básico para E/S con formato utiliza el tipo `String`, que se estudia en la Sección 2.3. En la salida, `+` combina dos valores de tipo `String`. Si el segundo argumento no es un valor de tipo `String`, se crea un valor temporal de tipo `String` para él, siempre que sea de tipo primitivo. Estas conversiones al tipo `String` se pueden definir también para objetos (Sección 3.4.3). Para la entrada, debemos asociar un objeto `BufferedReader` a `System.in`. Entonces se lee un valor de tipo `String` que se puede analizar sintácticamente. En la Sección 2.6 puede encontrarse una discusión más detallada de la E/S, incluyendo el tratamiento de ficheros con formato.

1.4 Operadores básicos

Esta sección describe algunos de los operadores disponibles en Java. Estos operadores se usan para formar *expresiones*. Una constante o una variable es una expresión por sí misma, así como las combinaciones de constantes y variables mediante operadores. Una expresión seguida de un punto y coma es una instrucción simple. En la Sección 1.5, examinaremos otros tipos de instrucciones que introducen operadores adicionales.

1.4.1 Operadores de asignación

En la Figura 1.3 se muestra un programa Java sencillo que ilustra sobre el uso de algunos operadores. El operador básico de *asignación* es el signo de igualdad. Por ejemplo, en la línea 16 se asigna a la variable `a` el valor de la variable `c` (que en ese punto es 6). Cambios posteriores en la variable `c` no afectan a `a`. Los operadores de asignación se pueden encadenar, como en `z=y=x=0`.

```
1  public class TestOperador
2  {
3      // Programa de prueba de los operadores básicos.
4      // La salida es la siguiente:
5      // 12 8 6
6      // 6 8 6
7      // 6 8 14
8      // 22 8 14
9      // 24 10 33
10
11     public static void main( String [ ] args )
12     {
13         int a = 12, b = 8, c = 6;
14
15         System.out.println( a + " " + b + " " + c );
16         a = c;
17         System.out.println( a + " " + b + " " + c );
18         c += b;
19         System.out.println( a + " " + b + " " + c );
20         a = b + c;
21         System.out.println( a + " " + b + " " + c );
22         a++;
23         ++b;
24         c = a++ + ++b;
25         System.out.println( a + " " + b + " " + c );
26     }
27 }
```

Figura 1.3 Programa que ilustra el uso de los operadores.

Java proporciona varios *operadores de asignación*, entre los que se encuentran =, +=, -=, *= y /=.

Otro operador de asignación es +=, cuyo uso se muestra en la línea 18 de la figura. El operador += añade el valor situado en el lado derecho (del operador de asignación +=) a la variable del lado izquierdo. Por tanto, en la figura, el valor de c pasa de ser 6, antes de la línea 18, a ser 14.

Java proporciona otros operadores de asignación como -=, *=, /=, que modifican el valor de la variable situada en el lado izquierdo del operador vía sustracción, multiplicación y división, respectivamente.

1.4.2 Operadores aritméticos binarios

Java proporciona varios operadores aritméticos binarios, entre los que se encuentran +, -, *, / y %.

En la línea 20 de la Figura 1.3 se muestra uno de los *operadores aritméticos binarios* típicos de todos los lenguajes de programación: el operador de suma (+). El operador + hace que los valores de b y c se sumen; b y c no sufren ningún cambio. El valor resultante se asigna a a. Otros operadores aritméticos típicos usados en Java son -, *, / y %, que se usan respectivamente para restar, multiplicar, dividir y obtener el resto de una división entera. La división entera devuelve solamente la parte entera y descarta el resto.

Como es habitual, la suma y la resta tienen la misma precedencia, y dicha precedencia es menor que la del grupo formado por los operadores de multiplicación, división y resto; en consecuencia 1+2*3 se evalúa a 7. Todos estos operadores asocian de izquierda a derecha (luego 3-2-2 se evalúa a -1). Todos los operadores tienen precedencia y asociatividad. La tabla completa de operadores se encuentra en el Apéndice B.

1.4.3 Operadores unarios

Se definen varios *operadores unarios*, entre los que se encuentra -.

Además de los operadores aritméticos binarios, que requieren dos operandos, Java proporciona *operadores unarios*, que requieren un solo operando. El más familiar de estos operadores es el menos unario, que se utiliza para cambiar el signo del operando. Así, `-x` devuelve el valor de `x` cambiado de signo.

El *autoincremento* y *autodecremento* suman y restan 1 respectivamente. Los operadores mediante los que se realizan son `++` y `--`. Hay dos formas de incrementar y decrementar: prefija y postfija.

Java proporciona también el operador de autoincremento, para añadir 1 a una variable, denotado por `++`, y el operador de autodecremento, para restar 1 a una variable, denotado por `--`. El uso más benigno de esta característica se muestra en las líneas 22 y 23 de la Figura 1.3. En ambas líneas, el *operador de autoincremento* `++` añade 1 al valor de la variable. En Java, sin embargo, un operador aplicado a una expresión produce una expresión que tiene un valor. Aunque se garantiza que la variable será incrementada antes de la ejecución de la siguiente instrucción, surge una pregunta: ¿cuál es el valor de la expresión con autoincremento si se usa dentro de una expresión mayor?

En este caso, la posición de `++` tiene una importancia crucial. La semántica de `++x` es que el valor de la expresión es el nuevo valor de `x`. Esto recibe el nombre de *incremento prefijo*. Por el contrario, `x++` hace que el valor de la expresión sea el valor original de `x`. Esto recibe el nombre de *incremento postfijo*. Esta propiedad se muestra en la línea 24 de la Figura 1.3. Tanto `a` como `b` se incrementan en 1, mientras que `c` se obtiene sumando el valor original de `a` con el valor incrementado de `b`.

1.4.4 Conversiones de tipo

El operador de *conversión de tipo* se usa para generar una entidad temporal de un tipo nuevo.

El *operador de conversión* se usa para generar una entidad temporal de un tipo nuevo. Considere, por ejemplo,

```
double cociente;
int x = 6;
int y = 10;
cociente = x / y;          // ¡Probablemente mal!
```

La primera operación es la división, y puesto que `x` e `y` son enteros, el resultado es su división entera, obteniéndose por tanto 0. El entero 0 se convierte entonces de forma implícita al tipo `double` para que pueda ser asignado a `cociente`. Pero nosotros pretendíamos asignar a `cociente` el valor 0,6. La solución es generar una variable temporal para `x` o para `y`, de forma que la división se lleve a cabo usando las reglas del tipo `double`. Esto se haría de la siguiente manera:

```
cociente = ( double ) x / y;
```

Observe que ni `x` ni `y` se modifican. Se crea una variable temporal sin nombre, cuyo valor se usa para efectuar la división. El operador de conversión de tipo tiene mayor precedencia que la división, luego la conversión de tipo se efectúa sobre el valor de `x` y después se lleva a cabo la división (en lugar de efectuarse la conversión tras la división de dos valores de tipo `int`).

1.5 Instrucciones condicionales

En esta sección se estudian instrucciones que afectan al flujo de control: instrucciones condicionales y bucles. Como consecuencia, se introducen nuevos operadores.

1.5.1 Operadores relacionales y de igualdad

El test básico que podemos realizar sobre los tipos primitivos es la comparación. Esta se lleva a cabo usando operadores de igualdad y desigualdad, así como los operadores relacionales (menor que, mayor que, etc.).

En Java, los operadores de igualdad son `==` y `!=`. Por ejemplo,

En Java, los *operadores de igualdad* son `==` y `!=`.

```
exprIzda == exprDcha
```

se evalúa a `true` si `exprIzda` y `exprDcha` son iguales; en caso contrario, se evalúa a `false`. Análogamente,

```
exprIzda != exprDcha
```

se evalúa a `true` si `exprIzda` y `exprDcha` son distintas; en caso contrario, se evalúa a `false`.

Los *operadores relacionales* son `<`, `<=`, `>` y `>=`. Tienen un significado natural para los tipos predefinidos. Los operadores relacionales tienen mayor precedencia que los operadores de igualdad. Ambos tienen menor precedencia que los operadores aritméticos pero mayor precedencia que los operadores de asignación, de modo que el uso de paréntesis no suele ser necesario. Todos estos operadores asocian de izquierda a derecha, pero este hecho no es de gran utilidad: en la expresión `a < b < 6`, por ejemplo, el primer `<` genera un valor de tipo `boolean` y el segundo es ilegal porque `<` no está definido para los valores de ese tipo. En la siguiente sección se describe la forma correcta de llevar a cabo esta comparación.

Los *operadores relacionales* son `<`, `<=`, `>` y `>=`.

1.5.2 Operadores lógicos

Java proporciona *operadores lógicos* que se usan para simular los conceptos Y, O y NO del álgebra de Boole. Éstos se conocen con el nombre de *conjunción*, *disyunción* y *negación* respectivamente, y los operadores correspondientes son `&&`, `||` y `!`. El test de la sección anterior se implementa adecuadamente como `a < b && b < 6`. La precedencia de la conjunción y la disyunción es lo suficientemente baja como para que no sea necesario utilizar paréntesis. El operador `&&` tiene mayor precedencia que el operador `||`, mientras que `!` se agrupa con los demás operadores unarios. Los argumentos y resultados de estos operadores lógicos son valores de tipo `boolean`. En la Figura 1.4 se muestran los resultados de aplicar los operadores lógicos para todos los posibles valores de entrada.

Java proporciona *operadores lógicos* que se usan para simular los conceptos Y, O y NO del álgebra de Boole. Los operadores correspondientes son `&&`, `||` y `!`.

X	Y	X && Y	X || Y	!X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Figura 1.4 Resultado de los operadores lógicos.

La *evaluación de ciclo corto* consiste en que cuando el resultado de un operador lógico se puede determinar examinando la primera expresión, entonces no se evalúa la segunda.

Una regla importante es que `&&` y `||` son operadores con evaluación de ciclo corto. La *evaluación de ciclo corto* consiste en que cuando el resultado de un operador lógico se puede determinar examinando la primera expresión, entonces no se evalúa la segunda. Por ejemplo, en

```
x != 0 && 1 / x != 3
```

si `x` es 0, entonces la primera mitad es `false`. Automáticamente el resultado del Y debe ser `false`, por lo que no se evalúa la segunda mitad. Esto es acertado pues la división por cero produciría un comportamiento erróneo. La evaluación de ciclo corto nos evita tener que preocuparnos por la división por cero[1].

1.5.3 La instrucción `if`

La instrucción `if` es la principal herramienta para tomar decisiones.

La instrucción `if` es la principal herramienta para tomar decisiones. Su forma básica es

```
if ( expresión )
    instrucción
siguiente instrucción
```

Si *expresión* toma el valor `true`, entonces se ejecuta *instrucción*; en caso contrario, no se ejecuta. Cuando se completa la ejecución de la instrucción `if` (sin ningún error sin tratar), el control pasa a la siguiente instrucción.

Opcionalmente, se puede usar una instrucción `if-else`, como la siguiente:

```
if( expresión )
    instrucción1
else
    instrucción2
siguiente instrucción
```

[1] Hay casos (extremadamente) raros en los que es preferible no usar la evaluación de ciclo corto. En tales casos, los operadores `&` y `|`, con argumentos de tipo `boolean`, garantizan que ambos argumentos se evalúan, incluso aunque el resultado de la operación pueda determinarse a partir del primer argumento.

En este caso, si *expresión* se evalúa a `true`, entonces se ejecuta *instrucción1*; en caso contrario se ejecuta *instrucción2*. En cualquier caso, el control pasa después a la siguiente instrucción, como en

```
System.out.print ( "1/x es " );
if( x != 0 )
    System.out.print( 1 / x );
else
    System.out.print( "Indefinido" );
System.out.println( );
```

Recuerde que se permite a lo sumo una instrucción en cada una de las cláusulas `if` y `else`, independientemente del sangrado. A continuación se puede observar dos errores:

```
if( x == 0 );     // ; es la instrucción vacía (y cuenta)
    System.out.print( "x es 0 " );
else
    System.out.print( "x es " );
    System.out.println( x ); // Dos instrucciones
```

El primer error es la inclusión del ; al final del primer `if`. Este punto y coma cuenta como la *instrucción vacía*; como consecuencia, este fragmento no podrá compilarse (ya no hay ningún `if` con el que se asocie `else`). Una vez arreglado ese error, tenemos un error lógico: la última línea no es parte del `else`, incluso aunque el sangrado lo sugiera. Para arreglar este problema, tenemos que usar un *bloque*, en el cual encerramos una secuencia de instrucciones entre un par de llaves:

Un punto y coma representa la *instrucción vacía*.

Un *bloque* es una secuencia de instrucciones entre llaves.

```
if( x == 0)
    System.out.println( "x es cero" );
else
{
    System.out.print( "x es " );
    System.out.println( x );
}
```

La instrucción `if` puede formar parte también de una cláusula `if` o `else`, al igual que otras instrucciones de control que estudiaremos más adelante en este capítulo. En el caso de instrucciones `if-else` anidadas, la cláusula `else` está asociada con la cláusula `if` más interna que esté sin emparejar. Será necesario añadir llaves en el caso de que no sea ése el significado pretendido.

1.5.4 La instrucción `while`

Java proporciona tres formas básicas de bucle: la instrucción `while`, la instrucción `for` y la instrucción `do`. La sintaxis de la *instrucción* `while` es la siguiente

La *instrucción* `while` es una de las tres formas básicas de bucle.

```
while( expresión )
    instrucción
siguiente instrucción
```

Observe que al igual que en la instrucción `if`, no hay punto y coma en la sintaxis. Si está presente, se interpreta como la instrucción vacía.

Mientras que *`expresión`* tiene el valor `true`, se ejecuta *`instrucción`*; en consecuencia, *`expresión`* está continuamente reevaluándose. Si expresión vale inicialmente `false`, *`instrucción`* no se ejecutará nunca. En general, *`instrucción`* lleva a cabo alguna acción que potencialmente puede alterar el valor de *`expresión`*; en caso contrario, el bucle podría ser infinito. Cuando el bucle `while` termina (con normalidad), el control pasa a la siguiente instrucción.

1.5.5 La instrucción `for`

La *instrucción* `for` es una construcción de bucle que se usa principalmente en iteración simple.

La instrucción `while` es suficiente para expresar todo tipo de repetición. Aun así, Java proporciona otras dos formas de bucle: la instrucción `for` y la instrucción `do`. La *instrucción* `for` se usa principalmente para la iteración simple. Su sintaxis es

```
for( inicialización; test; actualización )
     instrucción
siguiente instrucción
```

Aquí, *`inicialización`*, *`test`* y *`actualización`* son expresiones, siendo las tres opcionales. El valor por defecto de *`test`*, cuando no aparece, es `true`. No hay punto y coma tras el paréntesis cerrado.

Al ejecutar la instrucción `for`, lo primero que se lleva a cabo es la *`inicialización`*. Entonces, mientras *`test`* valga `true`, se llevan a cabo las siguientes acciones: se ejecuta *`instrucción`*, y se produce la *`actualización`*. Si se omiten *`inicialización`* y *`actualización`*, entonces la instrucción `for` se comporta exactamente igual a una instrucción `while`. La ventaja de la instrucción `for` es la claridad en el tratamiento de las variables contador (o iteradoras), ya que la instrucción `for` permite ver fácilmente cuál es el rango del contador. El siguiente fragmento produce los 100 primeros enteros positivos:

```
for( int i = 1; i <= 100; i++ )
    System.out.println( i );
```

Este fragmento ilustra la técnica habitual de declarar un contador en la parte de inicialización del bucle. El ámbito del contador es solamente el interior del bucle.

Tanto *`inicialización`* como *`actualización`* pueden usar una coma para permitir varias expresiones. El siguiente fragmento ilustra esta técnica habitualmente utilizada:

```
for( i = 0, suma = 0; i <= n; i++, suma += n )
    System.out.println( i + "\t" + suma );
```

Los bucles se anidan de la misma forma que las instrucciones `if`. Por ejemplo, podemos encontrar todos los números pequeños cuya suma es igual a su producto (como 2 y 2, cuya suma y producto es 4):

```
for( int i = 1; i <= 10; i++ )
    for( int j = 1; j <= 10; j++ )
        if( i + j == i * j )
            System.out.println( i + ", " + j );
```

Sin embargo, como veremos más adelante, cuando anidamos bucles podemos crear fácilmente programas cuyos tiempos de ejecución crecen rápidamente.

1.5.6 La instrucción do

La *instrucción* do es una construcción de bucle que garantiza que el cuerpo del bucle se ejecuta al menos una vez.

La instrucción `while` lleva a cabo un test repetidamente. Si el test se evalúa a `true`, entonces se ejecuta el cuerpo del bucle. Sin embargo, si el test tiene inicialmente el valor `false`, el cuerpo del bucle no se ejecuta nunca. Ahora bien, en algunos casos nos gustaría garantizar que el cuerpo del bucle se ejecuta al menos una vez. Esto se puede conseguir usando la instrucción `do`. La *instrucción `do`* es idéntica a la instrucción `while`, excepto por el hecho de que el test se realiza tras la ejecución del cuerpo del bucle. La sintaxis es

```
do
    instrucción
while( expresión );
siguiente instrucción
```

Observe que la instrucción `do` incluye un punto y coma. En el siguiente fragmento en pseudocódigo se muestra un uso típico de la instrucción `do`:

```
do
{
    Pedir dato al usuario;
    Leer valor;
} while( el valor no es bueno );
```

La instrucción `do` es la menos usada de las tres formas de bucle. Sin embargo, cuando tenemos que hacer algo al menos una vez, y por alguna razón un bucle `for` no es lo más adecuado, entonces la instrucción `do` se convierte en nuestra elección.

1.5.7 break y continue

Las instrucciones `for` y `while` ofrecen una oportunidad de terminación antes de volver a ejecutar el cuerpo del bucle. La instrucción `do` ofrece una oportunidad de terminación tras la ejecución del cuerpo. Ocasionalmente, podríamos querer terminar la ejecución en mitad del cuerpo (instrucción compuesta) del bucle. La *instrucción `break`*, formada por la palabra reservada `break` seguida de un punto y coma, se puede usar para conseguirlo. Normalmente una instrucción `if` precede a la instrucción `break` como en

```
while( ... )
{
    ...
    if( algo )
        break;
    ...
}
```

La *instrucción* `break` provoca la salida del bucle más interno o de la instrucción `switch` más interna. La *instrucción* `break` *etiquetada* provoca la salida de un bucle anidado.

La instrucción `break` provoca la salida del bucle más interno en el que está contenida (también se usa en conjunción con la instrucción `switch`, que se estudia en la siguiente sección). Si hay varios bucles de los que deseamos salir, la instrucción `break` no funcionará adecuadamente; además, lo más probable en tal caso es que su programa esté pobremente diseñado. Aun así, Java proporciona una instrucción *`break` etiquetada*. Para utilizar dicha instrucción, se comienza por poner una etiqueta a un bucle, de forma que la instrucción `break` se pueda aplicar a ese bucle, independientemente de cuantos bucles anidados haya. A continuación se muestra un ejemplo:

```
exterior:
  while( ... )
  {
    while( ... )
        if( desastre )
            break exterior; // Salta hasta después de exterior
  }
  // El control salta hasta aquí al salir del bucle exterior
```

La *instrucción* `continue` salta a la siguiente iteración del bucle más interno.

A veces, queremos abandonar la ejecución del cuerpo del bucle en la iteración actual y saltar a la siguiente iteración. Esto se puede realizar con la *instrucción `continue`*. Al igual que la instrucción `break`, la instrucción `continue` incluye un punto y coma y se aplica solamente al bucle más interno. El siguiente fragmento produce los 100 primeros enteros, con la excepción de los divisibles por 10:

```
for( int i = 1; i <= 100; i++ )
{
    if( i % 10 == 0 )
        continue;
    System.out.println( i );
}
```

Por supuesto, en este ejemplo hay otras alternativas al uso de la instrucción `continue`. No obstante, `continue` se usa a menudo para evitar patrones `if-else` complicados dentro de los bucles.

1.5.8 La instrucción `switch`

La *instrucción `switch`* se usa para elegir entre varios valores enteros pequeños. Consiste en una expresión y un bloque. El bloque contiene una secuencia de instrucciones y una colección de *etiquetas*, que representan posibles valores de la expresión. Todas las etiquetas deben ser distintas. Una etiqueta opcional por defecto

La *instrucción* `switch` se usa para elegir entre varios valores enteros pequeños.

encaja con cualquier valor no representado por las demás. Si no hay ningún caso aplicable al valor de la expresión `switch`, la instrucción `switch` finaliza su ejecución; en caso contrario, el control pasa a la etiqueta apropiada y se ejecutan todas las instrucciones desde ese punto en adelante. Se puede usar una instrucción `break` para forzar la terminación de la instrucción `switch` utilizándose casi siempre para separar lógicamente los distintos casos. En la Figura 1.5 se muestra un ejemplo de esta estructura típica.

```
1  switch( algunCaracter )
2  {
3    case '(':
4    case '[':
5    case '{':
6      // Código para procesar los símbolos de apertura
7      break;
8
9    case ')':
10   case ']':
11   case '}':
12     // Código para procesar los símbolos de cierre
13     break;
14
15   case '\n':
16     // Código para tratar el carácter de fin de línea
17     break;
18
19   default:
20     // Código para tratar otros casos
21     break;
22 }
```

Figura 1.5 Ejemplo de una instrucción `switch`.

1.5.9 El operador condicional

El *operador condicional* `?:` se usa como abreviatura de una instrucción `if-else` simple. La forma general es

El operador condicional `?:` se usa como abreviatura de una instrucción `if-else` simple.

```
exprTest ? exprSi : exprNo
```

Primero se evalúa *exprTest*, seguida por *exprSi* o *exprNo*, produciendo así el resultado de la expresión completa. Si *exprTest* se evalúa a `true` entonces se evalúa *exprSi*; en caso contrario, se evalúa *exprNo*. La precedencia del operador condicional se encuentra justo por encima de la de los operadores de asignación. Esto nos evita tener que usar paréntesis cuando asignamos a una variable el resultado de un operador condicional. Como ejemplo, para asignar el mínimo entre `x` e `y` a la variable `valMin` haríamos lo siguiente:

```
valMin = x <= y ? x : y;
```

1.6 Métodos

Un *método* es similar a lo que es una función en otros lenguajes. La *cabecera de un método* consiste en un nombre, el tipo del resultado y la lista de parámetros. La *declaración del método* incluye también su cuerpo. Un método declarado `public static` es equivalente a una función global en C. En el *paso de parámetros por valor*, se copian los argumentos actuales en los parámetros formales. Las variables se pasan usando paso de parámetros por valor.

Lo que en otros lenguajes se conoce como función o procedimiento, en Java se llama *método*. En el Capítulo 3 se proporciona un tratamiento más completo de los métodos. Esta sección presenta algunos conceptos básicos para escribir funciones del estilo de C, como `main`, de forma que podamos escribir algunos programas simples.

La *cabecera de un método* consiste en un nombre, una lista (posiblemente vacía) de parámetros y el tipo del resultado. El código que realmente implementa el método, a veces llamado el cuerpo del método, es formalmente un bloque. La *declaración de un método* está formada por una cabecera y un cuerpo. En la Figura 1.6 se muestra un ejemplo de un método y una rutina `main` que lo invoca.

Colocando delante de un método las palabras `public static`, podemos imitar una función global del estilo de C. Aunque en algunos ejemplos es una técnica útil, no debe sobreutilizarse.

El nombre del método es un identificador. La lista de parámetros consiste en cero o más *parámetros formales*, cada uno de ellos con un tipo. Cuando se llama a un método, los *argumentos reales* se pasan a los parámetros formales usando asignación normal. Esto significa que los tipos primitivos se pasan siempre usando *paso de parámetros por valor*. Los argumentos reales no pueden ser modificados por la función. Como en la mayoría de los lenguajes de programación modernos, las declaraciones de los métodos pueden aparecer en cualquier orden.

```
1  public class MinTest
2  {
3      public static void main( String [ ] args )
4      {
5          int a = 3;
6          int b = 7;
7
8          System.out.println( min( a, b ) );
9      }
10
11     // Definición de función
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

Figura 1.6 Declaración y llamada de un método.

La *instrucción* `return` se usa para devolver un valor al punto de llamada. Si el tipo del resultado es `void`, entonces no se devuelve ningún valor, escribiéndose `return;` en tal caso.

La *instrucción* `return` se usa para devolver un valor al punto de llamada.

1.6.1 Sobrecarga de los nombres de los métodos

Supongamos que necesitamos escribir una rutina que devuelva el máximo de tres valores de tipo `int`. Una cabecera razonable para el método sería

```
int max( int a, int b, int c )
```

En algunos lenguajes, esto puede ser inaceptable si `max` se ha declarado previamente. Por ejemplo, podríamos tener también

```
int max( int a, int b )
```

Java permite la *sobrecarga* de nombres de los métodos. Esto significa que varios métodos pueden tener el mismo nombre y declararse en el mismo ámbito de clase mientras sus *signaturas* (es decir, los tipos de la lista de parámetros) sean diferentes. Cuando se produce una llamada a `max`, el compilador puede deducir cuál es el método que se pretende aplicar en base a la lista de argumentos reales. Dos signaturas pueden tener el mismo número de parámetros, siempre que el tipo de alguno de ellos sea distinto.

La *sobrecarga* de un nombre de método significa que varios métodos pueden tener el mismo nombre siempre que los tipos de la lista de parámetros sean distintos.

Observe que el tipo del resultado no está incluido en la signatura. Esto significa que no está permitido tener dos métodos dentro del mismo ámbito de clase cuya única diferencia sea el tipo del resultado. Métodos que se encuentren en ámbitos de clase distintos pueden tener el mismo nombre, signatura e incluso tipo del resultado; esto se estudia en el Capítulo 3.

1.6.2 Clases de almacenamiento

Las entidades declaradas dentro del cuerpo de un método son variables locales y solamente se puede acceder a ellas a través de su nombre, dentro del cuerpo del método. Dichas entidades se crean cuando se ejecuta el cuerpo del método y desaparecen cuando el cuerpo del método termina.

Una variable declarada fuera del cuerpo de un método es global a la clase. Es similar a las variables globales en otros lenguajes si se usa la palabra `static` (lo cual es bastante probable si se desea permitir el acceso a la entidad por parte de métodos estáticos). Si se usan las palabras `static` y `final`, se trata de constantes simbólicas globales. Por ejemplo,

Las variables declaradas `static final` son en realidad constantes.

```
static final double PI = 3.1415926535897932;
```

Observe la convención habitual de usar nombres en mayúsculas para los nombres de constantes simbólicas. Si el nombre está formado por varias palabras, se separan mediante el carácter de subrayado, como en `MAX_INT_VALUE`.

Si se omite la palabra `static`, entonces la variable (o constante) tiene un significado diferente, que se discute en la Sección 3.4.5.

Resumen

En este capítulo se han estudiado las características básicas de Java, tales como los tipos primitivos, los operadores, las instrucciones condicionales, los bucles y los métodos, las cuales se encuentran prácticamente en cualquier lenguaje.

Cualquier programa no trivial requerirá el uso de tipos no primitivos, llamados *tipos referencia*, de los que se habla en el siguiente capítulo.

Elementos del juego

bloque Secuencia de instrucciones entre llaves.

cabecera de un método Consiste en el nombre, tipo del resultado y lista de parámetros.

código-j Código intermedio portable generado por el compilador de Java.

comentarios Hacen el código más legible pero no tienen significado semántico. Java admite tres formas de comentarios.

constante de tipo cadena Constante consistente en una secuencia de caracteres encerrados entre dobles comillas.

constantes enteras en octal y hexadecimal Las constantes enteras se pueden representar en notación decimal, octal o hexadecimal. La notación octal se indica añadiendo un `0` delante del número; la hexadecimal se indica añadiendo `0x` o `0X` delante del número.

declaración de un método Consiste en una cabecera y un cuerpo.

entidad `static final` Constante global.

entrada estándar Se trata del teclado, a menos que se redirija. También hay canales para la salida estándar y el error estándar.

evaluación de ciclo corto Proceso por el cual, si el resultado de un operador lógico se puede determinar a partir del primer operando, entonces no se evalúa el segundo operando.

identificador Usado para nombrar una variable o un método.

instrucción `break` Instrucción que provoca la salida del bucle más interno o de la instrucción `switch` más interna.

instrucción `break` etiquetada Instrucción `break` usada para salir de bucles anidados.

instrucción `continue` Instrucción que salta a la siguiente iteración del bucle más interno.

instrucción `do` Construcción de bucle que garantiza que el cuerpo del bucle se ejecuta al menos una vez.

instrucción `for` Construcción de bucle usada principalmente para la iteración simple.

instrucción `if` La construcción principal para tomar decisiones.

instrucción `return` Instrucción usada para devolver información al punto de llamada.

instrucción `switch` Instrucción usada para elegir entre varios enteros pequeños.

instrucción vacía Instrucción que consiste en un punto y coma.

instrucción `while` La forma más básica de bucle.

java Intérprete de Java.

javac Compilador de Java; genera código-j.

`main` Función especial que se invoca al ejecutarse el programa.

método El equivalente en Java a una función.

método estático Método equivalente a una función global.

operador condicional (`?:`) Operador usado como abreviatura de instrucciones `if-else` simples.

operador de conversión de tipo Operador usado para generar una variable temporal sin nombre de un tipo nuevo.

operadores aritméticos binarios Se usan para realizar operaciones aritméticas básicas. Java proporciona varios, entre ellos `+`, `-`, `*`, `/` y `%`.

operadores de asignación En Java se usan para modificar el valor de una variable. Estos operadores incluyen `=`, `+=`, `-=`, `*=` y `/=`.

operadores de autoincremento (++) y autodecremento (--) Operadores que suman y restan 1, respectivamente. Hay dos formas de incrementar y decrementar: prefija y postfija.

operadores de igualdad En Java, `==` y `!=` se usan para comparar dos valores; devuelven `true` o `false` (según corresponda).

operadores lógicos Son `&&`, `||` y `!`, y se usan para simular los conceptos Y, O y NO del álgebra de Boole.

operadores relacionales En Java son `<`, `<=`, `>` y `>=`, y se usan para decidir cuál es el mayor o el menor de dos valores; devuelven `true` o `false`.

operadores unarios Requieren un operando. Se definen diversos operadores unarios, entre los que se encuentran el menos unario (`-`) y los operadores de autoincremento y autodecremento (`++` y `--`).

paso de parámetros por valor Mecanismo de paso de parámetros de Java en el que los argumentos reales se copian en los parámetros formales.

secuencia de escape Se usa para representar algunas constantes de tipo carácter.

signatura Combinación del nombre del método y los tipos de la lista de parámetros. El tipo del resultado no forma parte de la signatura.

sobrecarga del nombre de un método Consiste en permitir a varios métodos tener el mismo nombre mientras los tipos de la lista de parámetros sean distintos.

tipos enteros `byte`, `char`, `short`, `int` y `long`.

tipos primitivos En Java son los enteros, números en coma flotante, booleanos y caracteres.

Unicode Conjunto internacional de caracteres que contiene más de 30.000 caracteres distintos, los cuales cubren los principales lenguajes escritos.

Errores comunes

1. Añadir un punto y coma innecesario produce errores lógicos, pues el punto y coma representa la instrucción vacía. Esto significa que un punto y coma a continuación de una instrucción `for`, `while` o `if` puede pasar inadvertido y estropear el programa.
2. En tiempo de compilación, Java es capaz de detectar algunos casos en los que un método que se supone que devuelve un valor, falla al hacerlo. Pero en último lugar, es su responsabilidad el recordar devolver un valor.
3. Un `0` delante de un número convierte una constante entera en octal; luego `037` es equivalente al decimal `31`.
4. Usar `&&` y `||` para las operaciones lógicas; `&` y `|` no evalúan en ciclo corto.
5. La cláusula `else` se asocia a la cláusula `if` más interna sin emparejar. Es habitual olvidar incluir las llaves necesarias para asociar un `else` con un `if` alejado.
6. Cuando se usa una instrucción `switch`, es habitual olvidar la instrucción `break` entre los casos lógicos. Cuando esto sucede, el control pasa al siguiente caso; lo que, en general, no es el comportamiento deseado.
7. Las secuencias de escape comienzan con la barra inclinada hacia atrás `\`, no con la inclinada hacia delante `/`.

8. Llaves sin emparejar pueden dar lugar a respuestas erróneas o engañosas. Usar `comprobarEquilibrados`, descrito en la Sección 11.1, para comprobar si ésta es la causa de un mensaje de error del compilador.
9. El nombre del fichero que contiene el código fuente del programa en Java debe ser igual que el nombre de la clase que se está compilando.

En Internet

Indicamos a continuación los ficheros disponibles correspondientes a este capítulo. Todos ellos son autocontenidos, y no serán posteriormente utilizados en el texto. Los programas se encuentran en el directorio **Chapter01**.

FirstProgram.java	Nuestro primer programa, que aparece traducido en la Figura 1.1.
MinTest.java	Para ilustrar el uso de los métodos, que aparece traducido en la Figura 1.6.
OperatorTest.java	Demostración del uso de varios operadores. Aparece traducido en la Figura 1.3.

Ejercicios

Cuestiones breves

1.1. ¿Qué extensiones se usan para los ficheros fuente y compilados en Java?
1.2. Describa los tres tipos de comentarios usados en Java.
1.3. ¿Cuáles son los ocho tipos primitivos en Java?
1.4. ¿Cuál es la diferencia entre los operadores `*` y `*=`?
1.5. Explique las diferencias entre los operadores de incremento prefijo y postfijo.
1.6. Describa los tres tipos de bucle en Java.
1.7. Describa todos los usos de una instrucción `break`.
1.8. ¿Qué hace la instrucción `do`?
1.9. ¿Qué es la sobrecarga de métodos?
1.10. Describa el paso de parámetros por valor.

Problemas teóricos

1.11. Supongamos que `b` tiene el valor 5 y `c` el valor 8. ¿Cuál es el valor de `a`, `b` y `c` después de la ejecución de cada una de las líneas del siguiente fragmento de programa?

```
a = b++ + c++;
a = b++ + ++c;
a = ++b + c++;
a = ++b + ++c;
```

1.12. ¿Cuál es el resultado de `true && false || true`?

1.13. Busque un ejemplo en el que el bucle `for` de la izquierda no sea equivalente al bucle `while` de la derecha:

```
for( inic; test; actual)
{
    instrucciones

}
```

```
inic;
while( test )
{
    instrucciones
    actual;
}
```

1.14. ¿Cuáles son las posibles salidas de este programa?

```
public class QueEsX
{
   public static void f(int x)
      { /* cuerpo desconocido */ }

   public static void main( String [ ] args )
   {
        int x = 0;
        f( x );
        System.out.println( x );
   }
}
```

Problemas prácticos

1.15. Escriba una instrucción `while` que sea equivalente al siguiente fragmento `for`. ¿Para qué podría ser esto útil?

```
for( ; ; )
    instrucción
```

1.16. Escriba un programa para generar las tablas de suma y multiplicación de números de un sólo dígito (las tablas que los estudiantes de la escuela elemental están acostumbrados a memorizar).

1.17. Presente dos métodos estáticos, el primero de los cuales debe devolver el máximo de tres enteros, mientras que el segundo devolverá el máximo de cuatro enteros.

1.18. Escriba un método estático que tome un año como parámetro y devuelva `true` cuando el año sea bisiesto.

Prácticas de programación

1.19. Escriba un programa que produzca todos los pares de enteros positivos, (a, b), tales que $a < b < 1000$ y $(a^2 + b^2 + 1)/(ab)$ sea entero.

1.20. Escriba un método que dado un número entero, muestre su representación en números romanos. En particular, si el dato es `1998`, la salida sería `MCMLXLVIII`.

1.21. Supongamos que queremos mostrar números entre corchetes, con el siguiente formato: `[1] [2] [3]`, etc. Escriba un método que tome dos parámetros: `cuantos` y `longLinea`. El método debe mostrar los números de línea desde 1 hasta `cuantos` en el formato antes mencionado, pero no debe mostrar más de `longLinea` caracteres en cada línea. Además, no debe empezar con el correspondiente a un nuevo número a menos que pueda escribir en la misma línea el correspondiente.

1.22. En el siguiente puzzle aritmético, se asigna un dígito a cada una de las diferentes letras. Escriba un programa que encuentre todas las soluciones posibles, a continuación mostramos una de ellas.

```
  MARK    A=1 W=2 N=3 R=4 E=5        9147
+ ALLEN   L=6 K=7 I=8 M=9 S=0      + 16653
 -----                              -----
 WEISS                              25800
```

Bibliografía

Una parte del material en el estilo de C usado en este capítulo se ha tomado de [5]. La especificación completa del lenguaje Java se puede encontrar en [4]. En [3] se listan algunos paquetes de librerías y se proporcionan ejemplos completos. Algunos libros introductorios de Java son [1] y [2][2].

1. G. Cornell y C. S. Horstmann, *Core Java*, 3.ª ed., Prentice-Hall, Englewood Cliffs, NJ (1998).
2. J. Lewis y W. Loftus, *Java Software Solutions: Foundations of Program Design*, Addison-Wesley, Reading, Mass. (1997).
3. D. Flanagan, *Java in a Nutshell*, 2.ª ed., O'Reilly and Associates, Sebastopol, Calif. (1997).
4. J. Gosling, B. Joy, y G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Mass. (1996).
5. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[2] La referencia [4] describe Java 1.0.2. En el momento en que se escribió la versión original de este libro, Java 1.1 ya se había introducido. La misma incorpora algunos cambios significativos. En una posible segunda edición de este texto se utilizarían aquellas características novedosas del lenguaje que puedan ser de utilidad para mejorar la presentación en algunos puntos. Las referencias [1], [2] y [3] usan Java 1.1.