PROGRAMACIÓN

Desarrollo de Aplicaciones Multiplataforma

Manual del módulo

UD 1 – CONCEPTOS BÁSICOS DE LA PROGRAMACIÓN

Ander González Ortiz

I. Informático de Gestión y Sistemas de Información

1.1. INTRODUCCIÓN

En nuestra vida cotidiana, interactuamos con diversas máquinas que simplifican nuestras tareas a través de **interfaces hombre-máquina**. Estas interfaces nos permiten comunicarnos con las máquinas utilizando botones, ruletas o teclas. Por ejemplo, en un horno simple, podemos ajustar la temperatura y el tiempo de cocción mediante ruedas selectoras. Aunque internamente el horno funciona con corriente eléctrica y resistencias para generar calor, la interfaz oculta estos detalles y nos brinda una forma más sencilla de utilizarlo. (Figura 1.1).

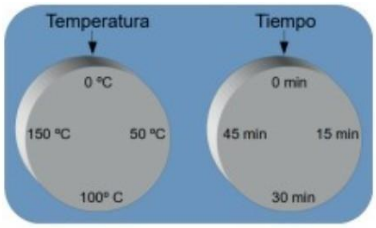


Figura 1. *Interfaz hombre-máquina para un horno simple.*

De manera similar, un ordenador es una máquina mucho más compleja en la que su funcionamiento interno se basa en voltajes que representan ceros y unos en un sistema binario. Aunque ejecutamos aplicaciones como videojuegos o procesadores de textos, en realidad, se trata de la manipulación de ceros y unos por componentes electrónicos. Sin embargo, programar en código binario resulta complicado y propenso a errores. Para superar este desafío, se utiliza una interfaz hombre-máquina llamada **lenguaje de programación**.

Así como la interfaz en el horno simplifica su uso, el lenguaje de programación facilita la interacción con un ordenador al abstraer la complejidad del código binario. En lugar de memorizar largas secuencias de ceros y unos, podemos utilizar instrucciones más comprensibles para realizar tareas como sumas o multiplicaciones. La interfaz hombre máquina en forma de lenguaje de programación nos permite programar de manera más eficiente y evitar errores costosos.

1.2. ALGORITMO

Podemos definir un algoritmo como un conjunto finito de instrucciones bien definidas que nos ayudan a resolver un problema. El algoritmo para preparar una pizza, que en el mundo gastronómico se conoce como receta, es:

1. Introducir la pizza en el horno.
2. Colocar la temperatura a 150ºC.
3. Colocar el tiempo a 15 min.
4. Esperar.
5. Retirar y comer.

Estamos acostumbrados a utilizar multitud de algoritmos, que son los procedimientos que realizamos de forma mecánica para solucionar un problema. Algunos ejemplos son: recetas de cocina, procesos para realizar operaciones matemáticas (sumas, multiplicaciones, etc.), pulsar los botones adecuados y en el orden correcto para que cualquier máquina haga su trabajo, etcétera.

Veamos el algoritmo para sumar dos números, utilizando a modo de ejemplo los números 2616 y 3708:

1. Colocar ambos números en dos filas haciendo coincidir las cifras del mismo orden (unidades con unidades, decenas con decenas y así sucesivamente) dos a dos.

```
  2 6 1 6
+ 3 7 0 8
---------
```

2. Comenzar por la derecha.

3. Hacer la suma de un solo guarismo de cada operando, anotando debajo las unidades resultantes y en la parte superior del guarismo de la izquierda las decenas, si existieran.

```
    1
  2 6 1 6
+ 3 7 0 8
---------
      4
```

4. Repetir el punto 3 con el guarismo de la izquierda.

```
    1
  2 6 1 6
+ 3 7 0 8
---------
    2 4
```

5. Terminar cuando no queden más elementos por sumar.

```
  1   1
  2 6 1 6
+ 3 7 0 8
---------
  6 3 2 4
```

1.3. LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación puede definirse como un idioma artificial diseñado para que sea fácilmente entendible por un humano e interpretable por una máquina. Consta de una serie de reglas y de un conjunto de órdenes o instrucciones. Cada una de estas instrucciones realiza una tarea determinada. A través de una secuencia de instrucciones podemos indicar a una computadora el algoritmo que debe seguir para solucionar un problema dado. A un

algoritmo escrito utilizando las instrucciones de un lenguaje de programación se le denomina *programa*.

Existen multitud de lenguajes de programación, cada uno con sus ventajas e inconvenientes. Disponemos de lenguajes especializados para realizar cálculos científicos, para escribir videojuegos o para programar robots.

Entre todos los lenguajes hemos elegido Java por ser de propósito general, sencillo y didáctico, sin dejar de ser potente y escalable. Quizá, junto al lenguaje C, sea el lenguaje de programación más utilizado por empresas e instituciones científicas y académicas.

1.4. EL PROGRAMA PRINCIPAL

Cuando se aprende a programar, durante la primera etapa, es posible que la frase, «Esto requiere unos conocimientos que están fuera del alcance de un principiante» aparezca demasiadas veces. Pero aquí está plenamente justificada; más adelante veremos los conceptos de clase y función, pero, por ahora, para escribir los primeros programas, utilizaremos:

```
package miprimerprograma; public
class Main {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		algoritmo

	}
}
```

Al escribir un programa en Java usaremos literalmente la fórmula anterior, aunque todavía no la comprendamos. De hecho, ni siquiera tendremos que escribir nada, ya que Elipse la escribirá por nosotros. Solo tenemos que sustituir algoritmo por el conjunto de instrucciones que necesitemos.

Hay que destacar que la primera línea de código, especifica que nuestro programa se agrupará en el paquete *miprimerprograma*. El nombre del paquete dependerá del nombre del proyecto; dicho de otra forma, Eclipse escribirá un nombre distinto de paquete dependiendo del nombre que se asigne a los proyectos.

Por este motivo, en la solución de los ejercicios hemos omitido la línea con la sentencia *package*.

1.5. PALABRAS RESERVADAS

En Java existe una serie de palabras con un significado especial, como *package*, *class* o *public*. Estas se denominan palabras reservadas y definen la gramática del lenguaje. En la Figura 2, se muestra el conjunto de las palabras reservadas en Java.

abstract	continue	float	native	strictfp	void
assert	default	for	new	super	volatile
boolean	do	if	package	switch	while
break	double	implements	private	synchronized	yield
byte	else	import	protected	this	
case	enum	instanceof	public	throw	
catch	extends	int	return	throws	
char	final	interface	short	transient	
class	finally	long	static	try	

Figura 2. *Palabras reservadas Java*.

Al conjunto anterior hay que sumar dos palabras reservadas muy curiosas: *const* y *goto*, que no pueden utilizarse en el lenguaje, pero aun así están reservadas. Además, existen tres valores literales: *true*, *false* y *null*, que tienen también un significado especial para el lenguaje, con un estatus parecido a una palabra reservada.

Las palabras reservadas solo pueden escribirse en determinado lugar de un programa y no pueden ser utilizadas como identificadores.

1.6. CONCEPTO DE VARIABLE

La Real Academia de la Lengua Española define variable como la magnitud que puede tener un valor cualquiera de los comprendidos en un conjunto. Dicho con otras palabras: una variable es una representación, mediante un identificador, de un valor, que puede cambiar durante la ejecución de un programa. A las variables se les asignan valores concretos por medio del operador de asignación (=). Ejemplo de ello es:

a=3

Aquí el nombre o identificador de la variable es a, y el valor asignado es 3. Esto no significa que posteriormente no pueda cambiar su valor por otro. Otro ejemplo:

a= 10 b=a+1

Utilizamos dos variables a y b. En la primera asignación damos un valor de 10 a la variable a, y en la segunda as ignición damos a b el valor que tuviera a más 1. Como a vale 10, b tomará un valor de 10 más 1, es decir, 11.

1.6.1. Identificadores

El nombre con el que se identifica cada variable se denomina identificador. Hay que tener en cuenta que Java distingue entre mayúsculas y minúsculas, es decir, el identificador edad es distinto a eDaD. Además, no podemos utilizar como identificador ninguna palabra reservada del lenguaje. Los identificadores deben seguir las siguientes reglas:

- Comienzan siempre por una letra, un subrayado (_) o un dólar ($).
- Los siguientes caracteres pueden ser letras, dígitos, subrayado (_) o dólar ($).
- Se hace distinción entre mayúsculas y minúsculas.
- No hay una longitud máxima para el identificador.

Existe una regla de estilo que recomienda distinguir las palabras que forman un identificador escribiendo en mayúscula la primera letra de cada palabra. Esta notación hace que el aspecto del identificador se asemeje a las jorobas de un camello, de ahí su nombre: notación Camel. Algunos ejemplos de identificadores que usen la notación Camel son los siguientes: edad, maxValor, numCasasLocalidad o notaMediaTercerTrimestre.

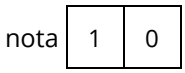
1.7. TIPOS PRIMITIVOS

En un programa en ejecución, las variables se almacenan en la memoria del ordenador. Cada una de ellas necesita un tamaño para guardar sus valores. Un tamaño demasiado pequeño no permite guardar valores grandes o muy precisos, y se corre el riesgo de que el valor que se va a guardar no quepa en el espacio reservado. Por el contrario, utilizar un tamaño excesivamente grande desaprovecha la memoria, haciendo un uso ineficiente de ella.

Veamos un ejemplo: la variable nota, que utilizaremos para guardar las calificaciones de los alumnos, almacenará valores que están comprendidos en el rango de 0 a 10. Con un tamaño en memoria para dos dígitos es suficiente.

```
nota= 10
```

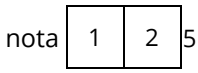
La forma de guardar esta información en la memoria es:



Por el contrario, si deseamos utilizar calificaciones comprendidas entre O y 300, dos dígitos no son suficientes.

```
nota= 125
```

Asignar 125 a la variable nota hace que el valor no pueda guardarse en el espacio reservado (que solo son dos dígitos):



Lo ideal sería que cada variable reserve un espacio lo suficientemente grande para que pueda almacenar todos los valores que guardará en algún momento, pero esto no siempre es posible.

La solución a este problema no es definir un tamaño de memoria para cada variable, sino definir unos tipos de variables, con unos tamaños y rangos de valores conocidos, y que las variables utilizadas en nuestros programas se ciñan a estos tipos.

En Java encontramos los tipos predefinidos: byte, short, int, long, float, double, boolean y char, también conocidos como tipos primitivos. Estos tienen un tamaño predefinido y pueden almacenar valores dentro de unos rangos (a mayor tamaño de memoria, mayor es el rango de posibles valores). Si con estos tipos primitivos no cubrimos nuestras necesidades, en unidades posteriores veremos cómo crear otros.

1.7.1 Variables de tipo primitivo

Al escribir un programa, hemos de indicar a qué tipo pertenece cada variable. Este proceso recibe el nombre de **declaración de variables** y se hará forzosamente antes de su primer uso. Veamos como ejemplo la forma de declarar la variable importe de tipo double:

```
double importe;
```

Todas las declaraciones de variables terminan en punto y coma (;), aunque es posible declarar a la vez varias del mismo tipo, separándolas por comas (,):

```
double importe, total, suma;
```

Existe la posibilidad de asignar un valor - inicializar- una variable en el momento de declararla,

```
double importe = 100.75;
```

que declara la variable importe de tipo double y le asigna un valor de 100,75.

Las variables de tipo primitivo a las que no se les asigna un valor en su declaración se inicializan automáticamente por defecto, de la siguiente forma: 0 para los tipos numéricos y char; y false para las variables booleanas. Aunque, por seguridad, Java no permite usarlos hasta que el usuario los inicialice.

1.7.2. Rangos

A continuación, se describe el tamaño (espacio que ocupa en memoria) y el rango de valores que puede almacenar cada tipo primitivo:

Tipo	Uso	Tamaño	Rango
byte	Entero corto	8 bits	de -128 a 127
short	Entero	16 bits	de -32 768 a 32767
int	Entero	32 bits	de -2147483648 a 2147483647
long	Entero largo	32 bits	± 9223372036854775808
float	Real precisión sencilla	32 bits	de -10^{32} a 10^{32}
double	Real precisión double	64 bits	de -10^{300} a 10^{300}
boolean	Lógico	64 bits	`true` o `false`
char	Carácter	1 bit	Cualquier carácter
String	Texto	16 bits	Cualquier texto

El desbordamiento de memoria puede ocurrir cuando un dato ocupa más espacio del asignado. El espacio extra se tomaría de la memoria adyacente, ocupándola. Y es aquí donde aparece el verdadero problema: desconocemos qué es lo que había almacenado en la porción extra de memoria que hemos sobrescrito. Quizá tengamos la suerte de que esté vacío o, por el contrario, podríamos estar destruyendo algún dato crucial. En Java no existe el desbordamiento de memoria, al disponer el lenguaje de un fuerte control de tipos que impide que se puedan realizar operaciones con desbordamiento. Sin embargo, sí existen lenguajes donde el control de tipos es menos exhaustivo o incluso inexistente, y donde sí podemos encontrarnos con situaciones de desbordamiento de memoria.

1.8. CONSTANTES

Las constantes son un caso especial de variables, donde, una vez que se les asigna su primer valor, este permanece inmutable durante el resto del programa. Cualquier dato que no cambie es candidato a guardarse en una constante. Ejemplos de constante son el número π, el número e o el IVA aplicable.

La declaración de constantes es similar a la de variables, pero añadiendo la palabra reservada final:

```
final tipo nombreConstante;
```

La mayoría de los programadores suele escribir sus códigos siguiendo una guía de estilos. Es habitual que los identificadores de las constantes se escriban en mayúscula. Nada nos impide escribirlas de otra forma, pero se hace así para distinguirlas, de un solo vistazo, de las variables. Un ejemplo de la declaración de constantes es el siguiente:

```
final int MAYORIA_EDAD = 18; final
double PI= 3.141592;
```

También es posible declarar la constante y posteriormente asignarle su valor:

```
final int NUM_ALUMNOS;
…
NUM_ALUMNOS = aulas * 30; //el valor es fruto de una expresión
```

Una vez que se ha asignado el valor a una constante, si intentamos modificarla, se producirá un error.

1.9. COMENTARIOS

Un programa no solo está formado por instrucciones del lenguaje; también es posible incluir notas o comentarios. El objetivo de estos es doble: describir la funcionalidad del código (qué hace) y facilitar la comprensión de la solución implementada (cómo lo hace).

Se considera una buena práctica escribir códigos bien documentados. Están especialmente indicados para facilitar el mantenimiento (modificación futura) de los programas: un código con el que trabajemos habitualmente y que conocemos con gran exactitud puede convertirse en un galimatías tras un tiempo sin trabajar con él; por otra parte, cuando se trabaja en colaboración con otros programadores, los comentarios que acompañan al código ayudan al resto del equipo.

Java dispone de tres tipos de comentarios:

- Comentario multilínea: cualquier texto incluido entre los caracteres / * (apertura de comentario) y */ (cierre de comentario) será interpretado como un comentario, y puede extenderse a través de varias líneas.
- Comentario hasta final de línea: todo lo que sigue a los caracteres // hasta el final de la línea se considera un comentario.

- Comentario de documentación: similar al comenta rio multilínea, con la diferencia de que, para iniciarlo, se utilizan los caracteres /**. Existen herramientas que generan documentación automática a partir de este tipo de comentarios.

Veamos algunos ejemplos:

```
/* esto es un comentario que se extiende durante varias líneas */
int numeroPaginas; //declararnos la variable declaramos como un entero

/** este comentario será utilizado en caso de utilizar una herramienta de
generación automática de documentación*/
```

1.10. API DE JAVA

Los lenguajes de programación modernos ofrecen una amplia biblioteca de herramientas que simplifican tareas complejas para los programadores. Si una herramienta específica no está disponible, el programador debe construirla, lo cual conlleva inconvenientes. Contar con herramientas ahorra tiempo y esfuerzo, además de proporcionar seguridad al ser desarrolladas y probadas por expertos. A estas herramientas, en Java, se les denomina clases y facilitan multitud de tareas. Algunos ejemplos de las funcionalidades que nos brindan son:

1. Lectura de datos: leen información desde el teclado, desde un fichero o desde otros dispositivos.
2. Cálculos complejos: realizan operaciones matemáticas como raíces cuadradas, logaritmos, cálculos trigonométricos, etcétera.
3. Manejo de errores: controlan la situación cuando se produce un error de algún tipo.
4. Escritura de datos: escriben información relevante en dispositivos de almacenamiento, impresoras, monitores, etcétera.

Estos son solo algunos ejemplos, pero la cantidad de clases que se distribuyen con Java es enorme y cubren las necesidades típicas de un programador. A toda esta biblioteca de clases se le denomina API, que son las siglas en inglés de «interfaz de programación de aplicaciones».

1.11. PAQUETES

El número de clases de la API es tal que, para facilitar su organización, se agrupan según su funcionalidad. A una agrupación de clases se le denomina paquete. Los paquetes pueden agruparse, a su vez, en otros paquetes. Por ejemplo, la clase Math, que proporciona herramientas para realizar cálculos matemáticos, se engloba dentro del paquete lang, que engloba clases que son fundamentales para el lenguaje, y que a su vez se encuentra dentro del paquete Java.

Cada clase se identifica mediante su nombre completo (o nombre cualificado) que incluye la estructura de paquetes junto al nombre de la clase. Por ejemplo, el nombre cualificado para la clase Math es: `java.lang.Math`.

Para utilizar cualquier clase de la API, tendremos que escribir su nombre cualificado. Por ejemplo, veamos cómo declarar una variable para guardar la hora. Para ello utilizaremos la clase LocalTime, que está ubicada en el paquete `java.time`,

```
java.time.LocalTime queHoraEs;
```

Igualmente, para usar cualquier método de una clase de la API, tendremos que escribir el nombre del método junto al nombre cualificado de la clase a la que pertenece. Por ejemplo, veamos cómo asignar a la variable queHoraEs la hora actual del sistema. Usaremos el método `now()` de la clase LocalTime,

```
java.time.LocalTime queHoraEs =  java.time LocalTime.now();
```

Tener que escribir continuamente el nombre cualificado de una clase (por ejemplo, `java.LocalTime`) puede llegar a ser engorroso. Una alternativa es declarar que vamos a utilizar una clase concreta, mediante la palabra reservada `import`. De la forma:

```
import java.time.LocalTime;
```

que se interpreta como: voy a necesitar "importar" en mi programa la clase LocalTime, que se encuentra dentro del paquete time, que a su vez se encuentra dentro del paquete java.

```
LocalTime queHoraEs = LocalTime.now(); //nombre corto
```

Es posible importar tantas clases como necesitemos y el hecho de importar una clase no nos obliga a utilizarla; tan solo acorta su escritura en la expresión donde la utilicemos. En ocas iones tener que importar una a una las clases de un paquete puede ser algo tedioso. Es posible importar todas las clases de un paquete mediante un asterisco:

```
import java.time.*; //importa todas las clases del paquete java.time
```

1.12. SALIDA POR PANTALLA

Una de las operaciones más básicas que proporciona la API es aquella que permite mostrar mensajes en el monitor, con idea de aportar información al usuario. Cuando los mensajes se muestran de forma simple, en modo texto y sin interfaz gráfica, se habla de salida por consola.

Java dispone para ello de la clase System con los métodos:

- `System.out.print("Mensaje")`, que muestra literalmente el mensaje en el monitor.
- `System.out.println("Mensaje")`, igual que el anterior pero, tras el mensaje, inserta un retorno de carro (nueva línea).

El uso de la clase System es tan básico que no es necesario importarla, esto lo hace Java por defecto. Incluso el entorno que vamos a utilizar para escribir programas (Eclipse) tiene un truco para escribir `System.out.println`: basta con escribir *`sout`* y pulsar la tecla tabulador. Eclipse lo escribirá por nosotros.

Para combinar la salida de mensajes literales de texto y el valor de las variables utilizaremos "+", que nos permite unir todos los elementos que deseemos para formar el mensaje de salida.

```
int edad= 12;
System.out.print("Su edad es de " + edad + " años.");
```

Se obtiene el mensaje en el monitor:

```
Su edad es de 12 años.
```

Lo que está entre comillas se muestra literalmente, mientras que edad, al no estar entrecomillado, se evalúa, mostrando su valor: 12.

1.13. ENTRADA DE DATOS

Otra operación muy utilizada, disponible en la API (mediante la clase Scanner), consiste en recaba información del usuario a través del teclado. Cuando se hace de forma sim- ple, en modo texto, sin ratón ni interfaz gráfica, se dice que obtenemos datos por consola.

Scanner es una clase de la API que se utiliza de forma no estática, es decir, necesita del operador new. Y la forma de trabajar con ella es siempre la misma: en primer lugar, tendremos que crear un nuevo escáner:

```
Scanner sc= new Scanner(System.in);
```

`System.in` indica que vamos a leer del teclado. Una vez creado nuestro escáner, que hemos llamado `sc`, ya solo queda utilizarlo. Para ello disponemos de los métodos:

1. `sc.nextInt()`: lee un número entero (`int`) por teclado.
2. `sc.nextDouble()`: lee un número real (double).
3. `sc.nextLine()`: lee una cadena de caracteres (una frase) hasta que se pulsa Intro.
4. `sc.next()`: lee una cadena de caracteres hasta que se encuentra un tabulador, un espacio en blanco o un Intro.

Las sentencias para introducir datos por teclado funcionan de la siguiente forma (Figura 3):

1. Se detiene la ejecución del programa y se espera a que el usuario teclee.
2. Recoge toda la secuencia tecleada hasta que se pulsa la tecla Intro.
3. Todo el contenido tecleado es interpretado y devuelto, normalmente asignándose a una variable.
4. El programa dispone del dato introducido por el usuario en la variable.

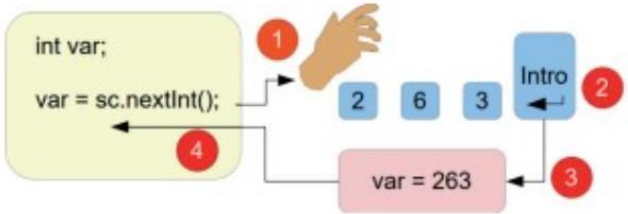


Figura 3. Entrada por consola. La clase Scanner interpreta la información tecleada por el usuario y la convierte en valores que son asignados a las variables.

Veamos un ejemplo completo de la lectura por consola de un número real:

```
Scanner sc = new Scanner(System.in); //crea el nuevo escáner
double numero; //declaramos la variable numero
numero= sc.nextDouble(); //se detiene la ejecución del programa hasta
//que escribimos un número en el área de Output y pulsamos Intro.
//ahora disponemos del valor introducido, mediante la variable numero
System.out.println("Ha escrito: " + numero);
```

Este fragmento de código se utilizará como una fórmula literal cada vez que necesitemos introducir información por teclado. Según deseemos leer un entero, un real o una cadena de caracteres, solo será necesario modificar el nombre y tipo de la variable que leer y el método de sc: nextlnt() , nextDouble( ), nextLine( ) o next().

Para utilizar la clase Scanner, como hemos hecho en el ejemplo anterior, sin tener que escribir su nombre cualificado, la importaremos de la siguiente forma:

```
import java.util.Scanner;
```

1.14. OPERACIONES BÁSICAS

Java dispone de multitud de operadores con los que se pueden crear expresiones utilizando como operandos variables, constantes, números y otras expresiones.

1.14.1. Operador de asignación

El operador = se usa para modificar el valor de una variable. La sintaxis es esta:

```
variable = expresión;
```

A la variable se le asigna como valor el resultado de la expresión. Una expresión no es más que una serie de operaciones. Si en el momento de la asignación la variable tuviera un valor anterior, este se pierde. Un ejemplo:

```
int total, a; //declaramos dos variables enteras
total= 123; //la variable total toma un valor de 123.
total = 0; //ahora toma un valor de 0. El valor 123 se pierde.
a= 3; //la variable a toma el valor 3
```

1.14.2. Operadores aritméticos

El operador - (menos unario) sirve para cambiar el signo de la expresión que le sigue, que estará formada por cualquier secuencia de operaciones aritméticas (Tabla 1 .2).

```
a = 1;
b = -1; // b vale -1
```

El operador % devuelve el resto de dividir el primer operando entre el segundo. Por ejemplo 7%3 (se lee 7 módulo 3) vale 1, ya que al dividir 7 entre 3 el resto (el módulo) es 1.

Símbolo	Descripción
+	Suma
+	Más unario: positivo
-	Resto
-	Menor unario: negativo
*	Multiplicación
/	División
%	Módulo
++	Incremento en 1
--	Decremento en 1

Los operadores ++ y -- se utilizan para incrementar o decrementar una variable en 1. El siguiente código:

```
a++;
b--;
```

es equivalente a:

```
a = a + 1;
b = b – 1;
```

1.14.3. Operadores relacionales

Son aquellos que producen un resultado lógico o booleano a partir de las comparaciones de expresiones numéricas (Tabla 1.3). El resultado solo permite dos posibles valores: verdadero o falso. En Java estos valores se representan mediante los literales true y false. Al principio, es usual confundir el operador de asignación (=) con el operador de comparación (==) y creemos estar comparando cuando en realidad estamos asignando. Comparemos de distintas formas los números 3 y 5:

1. 3 < 5. ¿Es 3 menor que 5? Es cierto; 3 es un número más pequeño que 5. Por tanto, la expresión devuelve true.
2. 3 == 5. ¿Es 3 igual que 5? Falso; ambos números son distintos, es decir, no son iguales. La expresión devuelve false.
3. 3 <= 5. ¿Es 3 menor o igual que 5? Cierto. La expresión devuelve true.
4. 3 <= 3. ¿Es 3 menor o igual que 3? Es cierto.
5. 3 != 4. ¿Es 3 distinto de 4? Cierto; ya que 3 es distinto a 4.

Símbolo	Descripción
--	Igual que
!=	Distinto que
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

1.14.4. Operadores lógicos

Permiten operar a partir de expresiones lógicas, formando expresiones más complejas, que devuelven, a su vez, un valor lógico (Tabla 1 .4). Existen los operadores and (conjunción Y), or (disyunción O) y not (negación).

Símbolo	Descripción
&&	Operador and: Y
||	Operador or: O
!	Operador not: negación

1.15.5 Operadores opera y asigna

Por simplicidad existen otros operadores de asignación llamados opera y asigna (Tabla 1 .5), que realizan la operación indicada tomando como operandos el valor de la variable a la izquierda y el valor a la derecha del =. El resultado se asigna a la misma variable utilizada como primer operando.

Símbolo	Descripción
+=	Suma y asigna
-=	Resta y asigna
*=	Multiplica y asigna
/=	Divide y asigna
%=	Módulo y asigna

Todos tienen el mismo funcionamiento. Utilizan la misma variable para operar con su valor y asignarle el resultado. Veamos a modo de ejemplo:

```
var += 3;
```

Lo que es equivalente a:

```
var = var + 3;
```

De igual forma,

```
x *= 2;
```

es equivalente a:

```
x = x* 2;
```

1.15.6. Operador ternario

Este operador devuelve un valor que se selecciona de entre dos posibles. La selección dependerá de la evaluación de una expresión relacional o lógica que, como hemos visto, puede tomar dos valores: verdadero o falso.

El operador tiene la siguiente sintaxis:

```
expresionCondicional ?  valor1 : valor2
```

La evaluación de la expresión decidirá cuál de los dos posibles valores se devuelve. En el caso de que la expresión resulte cierta, se devuelve valor1, y cuando la expresión resulte falsa, valor2.

Veamos un ejemplo:

```
int a, b;
a = 3 < 5 ? 1 : -1; // 3 < 5 es cierto: a toma el valor 1
b = a == 7 ? 10 : 20; // a (que vale 1) == 7 es falso: b toma valor 20
```

1.15.7. Precedencia

En la Tabla 1.6 se muestran los operadores ordenados, según su precedencia, de mayor a menor. En una expresión, la precedencia establece qué operaciones se realizan antes. Con igualdad de precedencia, las operaciones se realizan en el mismo orden en el que se escriben: de izquierda a derecha.

1. En la expresión: 2 + 3 * 4, el operador * tiene una precedencia mayor que el operador +, lo que significa que la multiplicación se realizará antes que la suma. Primero se hace la operación 3 * 4 (que es 12) y a continuación se realiza la suma 2 + 12 (que es 14).

2. En cambio, en la expresión: 3 <= 5 && 2 == 2, el operador con mayor precedencia es <=, que será la primera operación que se realice, siendo cierta. Queda:

   true && 2 == 2

   A continuación, se realizará la comparación, que también resulta cierta:

   true && true

Y el operador con menor precedencia de los tres es and lógico, que se realiza en último lugar, por lo que la expresión resulta cierta.

La precedencia puede romperse utilizando paréntesis; por ejemplo, en la expresión:

```
(2 + 3) * 4
```

el uso de paréntesis obliga a que la primera operación que se realice sea la suma.

1.16. CONVERSIÓN DE TIPOS

Como hemos visto, todas las variables en Java tienen asociado un tipo. Cuando asignamos un valor a una variable, ambos deben ser del mismo tipo. A una variable de tipo `int` se le puede asignar un valor `int` y a una variable `double` se le puede asignar un valor `double`.

```
int a = 2;
double x = 2.3;
```

Cada tipo se caracteriza por ocupar un tamaño en memoria, donde se almacenan los valores correspondientes.

Si escribimos

```
int a = 2.6; //trata de asignar un valor real a una variable entera
```

el compilador nos avisará de que estamos cometiendo un error y no nos dejará ejecutar. La razón de este control de tipos tan estricto es evitar errores durante la ejecución del programa, ya que es evidente que una variable de un tipo no puede almacenar valores con un tamaño superior. Por ejemplo, un valor double ocupa en la memoria 64 bits, mientras que una variable `int` utiliza 32 bits para almacenar un valor. Simplemente un valor `double` no cabe en una variable `int`.

Sin embargo, un valor `int` puede guardarse sin problemas en una variable double.

¿Por qué no permitir una asignación como esta?

```
int a= 3;
double x = a;
```

Java permite esta asignación sin violar la norma de que a una variable double se le asigne un valor double. Para ello, Java convierte de forma automática el valor entero 3 en el valor `double` 3.0 antes de asignarlo a la variable `x`. Esto es posible porque la variable double es de mayor tamaño que el valor `int`, es decir, tiene suficiente espacio para guardarlo. Por tanto, este tipo de conversiones y asignaciones automáticas será posible cuando la variable sea de mayor tamaño que el del valor asignado. Se habla entonces de conversiones de ensanchamiento. Nos permitirá, por ejemplo, guardar valores `byte`, `short`, `int`, `long` o `float` en una variable double.

Muy distinto es que intentemos asignar un valor double a una variable `int`, que no tiene sitio suficiente para guardarlo. Lo norma l es que se trate de un error del programador.

En este caso Java no hará ninguna conversión automática. Se limitará a darnos un error de compilación. Sin embargo, a veces es interesante guardar la parte entera de un número con decimales en una variable entera. Evidentemente, esto supone una pérdida de información, ya que los decimales desaparecerán. Para ello deberemos colocar un `cast` o `molde` delante del valor que queremos asignar,

```
int a = (int) 2.6; // (int) indica el tipo al que se convertirá el valor
```

El `cast` es lo que va entre paréntesis. Lo que hace es eliminar (truncar) la parte decimal de 2.6 y convertirlo en el entero 2, que podrá ser asignado a la variable a sin problemas.

Este tipo de conversión se llama de estrechamiento, ya que fuerza la asignación de un tipo de dato en una variable de menor tamaño, eso sí, con pérdida de información.

Nada impide, aunque no es necesario, colocar un `cast` en una conversión de ensanchamiento. Esto a veces hace que el programa gane en legibilidad:

```
double x = (double) 3;
```

1.17. CADENAS DE CARACTERES

Por texto entendemos una palabra, una frase e incluso uno o más párrafos de cualquier longitud. En definitiva, un texto, como por ejemplo este mismo párrafo, es una secuencia de caracteres, de ahí que también se le denomine cadena de caracteres o, por economía del lenguaje, simplemente cadena.

Para manipular textos disponemos en la API de las clases `Character` y `String` —ambas ubicadas en el paquete `java.lang`—, que proporcionan multitud de funcionalidades para trabajar con un solo carácter la primera y con textos de cualquier longitud la segunda.

1.17.1. Tipo primitivo char

De forma general un carácter se define como una letra —de cualquier alfabeto—, un número, un ideograma o cualquier símbolo. En Java un carácter o literal carácter se escribe entre comillas simples ('). Algunos ejemplos de ellos son: 'p', 'ñ', 'W' '7', '€' o '#'.

A la hora de seleccionar un carácter es posible usar su codificación Unicode o el propio carácter si es posible escribirlo mediante el teclado. Por ejemplo, el carácter 'a' puede escribirse pulsando la tecla adecuada del teclado o mediante su code point en decimal (97) o en hexadecimal (\u0061).

Veamos la forma de asignar 'a' a una variable de tipo `char`:

```
char c;
c = 'a'; //directamente mediante el teclado
c = 97; // usando el code point en decimal
c = '\u0061'; // o con el code point en hexadecimal
```

La única forma de designar el carácter '♥' es mediante su code point.

```
char c = '\u2661'; // o bien, c = 9825;
System.out.println(c); // muestra un ♥
```

1.17.2. Secuencias de escape

Un carácter precedido de una barra invertida (\) se conoce como secuencia de escape. Al igual que los caracteres escritos mediante la codificación Unicode, representan un único

carácter, pero en este caso poseen un significado especial. En la siguiente tabla se muestran las secuencias de escape de Java.

Símbolo	Descripción
\b	Borrado a la izquierda
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador
\f	Nueva página
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida

Algunos ejemplos:

```
char c = '\'';
System.out.println(c); // muestra una comilla simple:  '
c = '\"';
System.out.println (c); // muestra una comilla doble:  "
c = '\t'; // tabulador. Al ser invisible lo representamos con |________|
System.out.println("1" + c + "2"); // muestra 1_________2
```

1.17.3. La clase String

Las cadenas, conjuntos secuenciales de caracteres, se manipulan mediante la clase `String`, que funciona de forma dual. Por un lado, de manera general, tiene un funcionamiento no estático; pero a su vez, dispone de algunos métodos que sí lo son. Es posible definir variables de tipo `String` de la forma habitual:

```
String cad; // cad es una variable de tipo cadena
```

Una variable de tipo `String` almacenará una cadena de caracteres, que provendrá de la manipulación de otra cadena o de un literal. Una cadena literal consiste en un texto entre comillas dobles ("). Es posible utilizar cualquier carácter, incluidos los codificados mediante Unicode y las secuencias de escape. Algunos ejemplos de literal cadena son:

```
"Hola\n "
"En un lugar de la mancha"
"Un corazón: \u2661 "
```

Los literales carácter y cadena se diferencian en el tipo de comillas utilizado; mientras 'a' es un carácter, "a" es una cadena que está compuesta por un único carácter.

Con una cadena de caracteres se pueden realizar multitud de operaciones. Algunas trabajan con la cadena como un todo y otras trabajan carácter a carácter.

1.17.4. Comparación

Un error común es comparar dos variables de tipo cadena utilizando el operador de comparación. Este operador no se puede utilizar con `String` debido a que es una clase y no un tipo primitivo. Por ello, para comparar cadenas usaremos:

- `boolean equals (String otra)`: compara la cadena que invoca el método con otra. El resultado de la comparación se indica devolviendo true o false, según sean iguales o distintas. Para que las cadenas se consideren iguales deben estar formadas por la misma secuencia de caracteres, distinguiendo mayúsculas de minúsculas.

  Veamos un ejemplo:

```
String cad1 = "Hola mundo";
String cad2 = "Hola mundo";
String cad3 = "Hola, buenos días";
boolean iguales;
iguales = cad1.equals(cad2); // iguales vale true
iguales = cad1.equals(cad3); // iguales vale false
```

1.17.5. Concatenación

El operador **"+"** sirve para unir o concatenar dos cadenas. Veamos su funcionamiento con un ejemplo:

```
String nombre = "Miguel";
String apellidos = "de Cervantes Saavedra";
String nComp1eto = nombre + apellidos;
System.out.println(nComp1eto); // "Miguel de Cervantes Saavedra"
```

1.17.6. Obtención de una subcadena

Una subcadena es un fragmento de una cadena, es decir, un subconjunto de caracteres contiguos de una cadena. En ocasiones puede ser interesante extraer de una cadena un fragmento. Por ejemplo, si tenemos el nombre y los apellidos de alguien, puede ser útil extraer solo los apellidos. Los métodos que llevan a cabo esto son:

- `String substring (int inicio)`: devuelve la subcadena formada desde la posición inicio hasta el final de la cadena. Lo que se devuelve es una copia y la cadena invocante no se modifica.

```
String cad1 = "Una mañana, al despertar de un sueño intranquilo";
String cad2 = cad1.substring(28); //cad2 vale "un sueño intranquilo"
```

- String substring (int inicio, int fin): hace lo mismo que la anterior, devolviendo la subcadena comprendida entre los Índices inicio y el anterior a fin.

```
String cad1 = "Una mañana, al despertar de un sueño intranquilo";
String cad2 = cad1.substring (15, 36); // cad2 = "despertar de un
sueño"
```

Hay que notar que en cad1 el carácter que ocupa el índice 36 es el espacio en blanco, que va justo antes de intranquilo y que este carácter no forma parte de la subcadena devuelta. Esta se forma con los caracteres que se encuentran desde el Índice 15 hasta el carácter anterior al 36, es decir, el 35.

1.17.7. Longitud de una cadena

Como hemos visto, en ciertos métodos es necesario utilizar algunos Índices para localizar los caracteres que forman una cadena. Para evitar el uso de un índice que se encuentre fuera de rango, existe:

- `int length()`: devuelve el número de caracteres (longitud) de una cadena. Una vez conocida la longitud, podemos usar, sin miedo a generar un error, cualquier índice comprendido entre O y el índice del último carácter, que es la longitud de la cadena menos 1.

```
int longitud;
String cad1 = "Hola"
String cad2 = "";
longitud = cad1.length(); / / devuelve 4
longitud = cad2.length(); / / devuelve 0
```

1.17.8. Conversión

Una cadena puede transformarse sustituyendo todas las letras que la componen a minúsculas o a mayúsculas, lo que resulta útil a la hora de procesar, por ejemplo, valores que provienen de un formulario y que cada usuario puede escribir de una forma u otra.

Por homogeneidad se suele trabajar con todos los valores convertidos a un solo tipo de letra. Para realizar esta operación disponemos de:

- String `toLowerCase()`: devuelve una copia de la cadena donde se han convertido todas las letras a minúsculas.
- String `toUpperCase()`: similar al método `toLowerCase()`, convierte todas las letras a mayúsculas.

Solo se convierten las letras; el resto de caracteres se mantiene igual.

El método `replace()` permite sustituir todas las ocurrencias de un carácter de una cade- na por otro que se pasa como parámetro.

`String replace (char original, char otro)`: devuelve una copia de la cadena invocante donde se han sustituido todas las ocurrencias del carácter original por otro. Un ejemplo:

```
String frase = "Hola mundo";
frase = frase.replace('o' , 'O');
System.out.println(frase); // HOla mundO
// String replace(CharSequence original, CharSequence otra): cambia todas
las ocurrencias de la cadena original por la cadena otra.
```