



Conceptos básicos

Objetivos

- Establecer todos los conceptos básicos que son necesarios para el conocimiento de cualquier lenguaje de programación.
- Concretar la importancia de los mecanismos básicos que usan los compiladores, como la declaración y asignación de variables, el establecimiento de tipos, etcétera.
- Clasificar los distintos operadores, atendiendo a su uso y al resultado de la evaluación de sus expresiones.
- Profundizar en el concepto de tipo primitivo, sus tamaños, rangos y conversiones, así como sus valores literales.
- Asociar los tipos primitivos con sus operadores.
- Asimilar las distintas formas de interacción con el usuario mediante la salida por consola y la entrada de datos desde el teclado.
- Exponer el funcionamiento de herramientas (métodos) que proporciona la API para ciertas clases, la diferencia entre el uso estático y no estático de los métodos de la API, así como la clasificación de clases en los distintos paquetes.

Contenidos

- 1.1. Algoritmo
- 1.2. Lenguajes de programación
- 1.3. ¿Cuál es el propósito de este libro?
- 1.4. NetBeans IDE
- 1.5. El programa principal
- 1.6. Palabras reservadas
- 1.7. Concepto de *variable*
- 1.8. Tipos primitivos
- 1.9. Variables de objeto
- 1.10. Constantes
- 1.11. Comentarios
- 1.12. API de Java
- 1.13. Operaciones básicas
- 1.14. Conversión de tipos

Introducción

En nuestra vida cotidiana estamos en contacto con multitud de máquinas que, en la mayoría de los casos, simplifican nuestras tareas. La forma que tenemos de comunicarnos con ellas, a través de botones, ruletas o teclas, es lo que se conoce como *interfaz hombre-máquina*. Supongamos que queremos cocinar en nuestro horno. Lo ideal sería que el horno no tuviera ningún botón ni mando, solo un pequeño micrófono al que pudiéramos dirigirnos y expresarle nuestros deseos: «esta noche me apetece cenar una pizza de espinacas con extra de queso». Por desgracia, este tipo de interfaz todavía no se encuentra en nuestros electrodomésticos. Quizá en unos años.

Volviendo al presente, un horno sencillo donde solo podamos controlar la temperatura adecuada en cada momento y el tiempo que estaremos cocinando presenta los siguientes mandos (Figura 1.1).

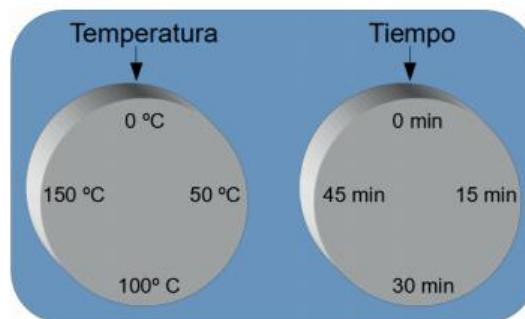


Figura 1.1. Interfaz hombre-máquina para un horno simple.

Girando ambas ruedas y colocándolas en la posición deseada podemos configurar o programar el horno. Aunque internamente un horno funciona dejando pasar corriente eléctrica a través de unas resistencias que generan calor, este proceso se interrumpe y continúa, a intervalos, para controlar la temperatura deseada mediante un termostato. Además, todo está dirigido por un cronómetro que se encarga de controlar el tiempo total de funcionamiento. La interfaz, nuestras dos ruedas selectoras de tiempo y temperatura, nos abstrae del verdadero funcionamiento interno del horno. No necesitamos tener ningún conocimiento sobre resistencias, termostatos ni cronómetros. Los mandos hacen invisible el funcionamiento interno y transforman el manejo en algo mucho más sencillo.

Veamos este mismo proceso para un ordenador, una máquina mucho más compleja que un horno. El funcionamiento interno de un ordenador depende de voltajes que cambian entre niveles bajos y altos; estos valores se representan también como ceros (0) y unos (1), un sistema de numeración binario, pues estos voltajes —o ceros y unos— que pasan a través de unos componentes electrónicos, son los que controlan el funcionamiento de la computadora. Cuando un videojuego o un procesador de textos se ejecutan en un ordenador, es difícil pensar que, en el fondo, no son más que el procesado de ceros y unos por dispositivos electrónicos. Para un humano, programar a través de ceros y unos es algo bastante complicado. Por ejemplo, indicarle a un ordenador que realice una suma se hace a través del código 0010101011011000 y una multiplicación es 0011000010101101.

Aquí se aprecia que, aparte de memorizar multitud de códigos binarios, un pequeño error puede ser un desastre y producir un resultado totalmente distinto del que pretendemos.

Para solucionar este problema, al igual que en nuestro horno, existe una interfaz hombre-máquina que nos facilita esta tarea. Esta interfaz se denomina *lenguaje de programación*.

1.1. Algoritmo

Continuando con nuestra cena, para cocinarla nos basta con seguir una serie de instrucciones y seleccionar la posición adecuada de cada mando en cada momento. Dicho de otra forma, tenemos que seguir un conjunto de pasos definidos para resolver el problema: ¿cómo cocinar una pizza?

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 150 °C.
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.

$$\begin{array}{r} 2 \ 6 \ 1 \ 6 \\ + \ 3 \ 7 \ 0 \ 8 \\ \hline \end{array}$$

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.

$$\begin{array}{r} & & & 1 \\ & 2 & 6 & 1 & 6 \\ + & 3 & 7 & 0 & 8 \\ \hline & & & & 4 \end{array}$$

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

$$\begin{array}{r} & & 1 \\ & 2 & 6 & 1 & 6 \\ + & 3 & 7 & 0 & 8 \\ \hline & 2 & 4 \end{array}$$

5. Terminar cuando no queden más elementos por sumar.

$$\begin{array}{r} & & 1 & 1 \\ & 2 & 6 & 1 & 6 \\ + & 3 & 7 & 0 & 8 \\ \hline & 6 & 3 & 2 & 4 \end{array}$$

1.2. 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. Por ejemplo, Fortran es un lenguaje de programación diseñado para realizar aplicaciones científicas. Podemos utilizarlo para calcular operaciones complejas fácilmente, pero sería tremadamente laborioso utilizarlo para escribir un videojuego. Igualmente existen lenguajes de propósito general —como por ejemplo el lenguaje C— que no están especializados en un campo concreto, pero con los que podemos realizar casi cualquier tarea con un mayor o menor esfuerzo.

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.

Argot técnico



Para conocer cuáles son los lenguajes más utilizados existen distintos sitios que realizan mediciones del número de proyectos que usan un lenguaje concreto.

Entre estos sitios destaca el **índice TIOBE**, que mide la popularidad de los lenguajes de programación. Esta se calcula a partir del número de resultados que proporcionan las consultas en los principales motores de búsqueda para un lenguaje de programación.

■ ■ ■ 1.2.1. Lenguajes compilados e interpretados

Un lenguaje de programación está diseñado para que una persona escriba fácilmente algoritmos, pero la circuitería de un ordenador no comprende ningún lenguaje distinto al sistema binario. ¿Cómo se consigue que un ordenador comprenda lo que se ha escrito mediante un lenguaje de programación? La solución es utilizar una herramienta llamada *compilador*, que transforma el conjunto de órdenes o instrucciones que escribimos utilizando cualquier lenguaje de programación —también llamado *código fuente*— en los ceros y unos que son comprensibles por la circuitería de la máquina, lo que se llama *código máquina* (véase Figura 1.2).

Con esta solución tan elegante podremos programar una máquina tan compleja como un ordenador, casi de la misma forma en la que utilizamos un horno, abstrayéndonos de su funcionamiento interno, sin conocimientos de electrónica y sin necesidad de entender todas y cada una de sus partes.

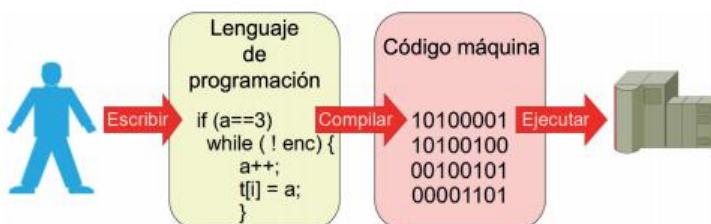


Figura 1.2. Interfaz hombre-máquina a través de un lenguaje de programación.

Existen dos enfoques para realizar el proceso de traducción del lenguaje de programación (*código fuente*) al *código máquina*. El primero, llamado *compilación*, que ya hemos visto, traduce todas las instrucciones del *código fuente* y almacena el *código máquina* generado. Esto permite ejecutar el programa, sin volver a compilarlo, tantas veces como se necesite y sin disponer del *código fuente*.

El otro enfoque se denomina *interpretación* y consiste en traducir el *código fuente* instrucción a instrucción e ir ejecutando, es decir, solo se traduce y ejecuta la siguiente instrucción que necesitamos. El proceso continuará sucesivamente hasta que la ejecución termine. Por regla general, el *código máquina* obtenido en la interpretación no se suele almacenar, lo que obliga a volver a interpretar cada vez que se necesite ejecutar un programa.

A partir del proceso de compilación e interpretación vamos a introducir dos nuevos conceptos:

- **Tiempo de compilación:** es el intervalo de tiempo durante el cual se compila un programa.
- **Tiempo de ejecución:** es el intervalo de tiempo durante el cual un programa se ejecuta.

De los distintos estadios por los que pasa un programa es importante entender en cuál de ellos estamos, ya que los tipos de comprobaciones y chequeos que se realizan dependen de la etapa en la que nos hallemos.

■ ■ ■ 1.2.2. Lenguajes multiplataforma

El lenguaje C sigue el esquema de la Figura 1.2, lo que necesita que un mismo programa se compile para cada combinación de tipo de máquina —IBM-PC, Macintosh, SPARC, etc.— y sistema operativo donde se va a ejecutar. Esto se debe a que el código máquina varía según los estándares hardware y software donde se va a ejecutar; estos definen el entorno de ejecución, también llamado *plataforma*.

Aunque se pueda ejecutar en distintas plataformas, el lenguaje C requiere del compilado específico para cada una de ellas. Por este motivo, no se considera un lenguaje multiplataforma.

Java se concibió como un lenguaje para internet y, por lo tanto, necesita ser multiplataforma, para que un mismo programa se compile una única vez y pueda ser ejecutado en multitud de ordenadores y sistemas operativos completamente diferentes. Esto se consigue añadiendo una capa extra a la solución representada en la Figura 1.2. El compilador de Java no genera un código máquina dependiente de ninguna plataforma; en su lugar, genera un código binario especial llamado *bytecode de Java*. Este no es ejecutable directamente por ningún ordenador, ya que es independiente de cualquier plataforma y ha sido ideado como un código intermedio por los implementadores de Java. Para poder ejecutarlo la solución está en disponer de un intérprete en cada equipo, que traduce el bytecode de Java al código máquina nativo de cada plataforma.

Al programa que interpreta el bytecode se le conoce como *Máquina Virtual de Java* o *JVM*, por sus siglas en inglés.

De esta forma, se consigue que Java sea un lenguaje multiplataforma: un mismo programa, una vez compilado, se puede ejecutar en cualquier ordenador que tenga instalada la Máquina Virtual de Java, que no es más que un intérprete de bytecode.

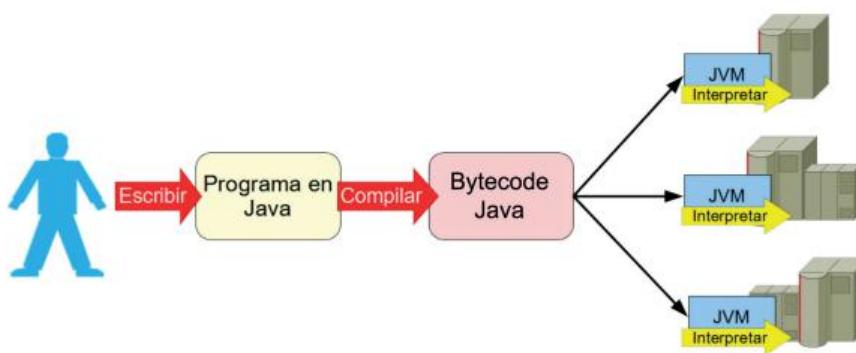


Figura 1.3. Mecanismos de compilación-interpretación para que Java sea un lenguaje multiplataforma y pueda ejecutar el mismo programa en distintos ordenadores instalados con la JVM.

Para el caso concreto de Java, vamos a afinar los conceptos de:

- **Tiempo de compilación:** es el espacio de tiempo en el que se traduce el código fuente al bytecode.
- **Tiempo de ejecución:** es el tiempo durante el cual el bytecode se interpreta (por la JVM) y se ejecuta por la plataforma correspondiente.

■ 1.3. ¿Cuál es el propósito de este libro?

El objetivo principal de este libro es doble. En primer lugar, que el lector conozca y aprenda el funcionamiento de las instrucciones o sentencias que proporciona Java y, en segundo lugar, que sea capaz de utilizarlas para escribir algoritmos correctos que resuelvan problemas reales. Estos pueden ser tan simples como calcular la suma de dos números o tan complejos como gestionar la parte financiera de una empresa o desarrollar un videojuego.

Aquí hay que hacer hincapié en que el conocimiento de Java no implica saber programar correctamente. Dicho de otro modo, conocer el funcionamiento individual de cada instrucción no garantiza el éxito; este se consigue teniendo una visión global del problema, conociendo y aplicando técnicas algorítmicas y escribiendo las instrucciones en el orden correcto.

■ 1.4. NetBeans IDE

Un programador dispone de multitud de herramientas para llevar a cabo su tarea. Lo más básico es un editor de texto donde escribir las instrucciones y un compilador que transforme el fichero de texto, con las sentencias de Java, en un fichero escrito en un lenguaje especial, capaz de ser interpretado por la Máquina Virtual de Java (JVM).

También hay entornos de programación más sofisticados que proporcionan una enorme cantidad de funcionalidades: editor de texto, ayuda, compilador, depurador y, en general, casi cualquier cosa que se nos pueda ocurrir. Estos entornos se conocen como IDE, las siglas en inglés de «entorno integrado de desarrollo», y son un conjunto de herramientas integradas orientadas al desarrollo de software.

De todos los entornos disponibles, hemos decidido utilizar NetBeans (Figura 1.4), que es gratuito y de código abierto. En la página web oficial de NetBeans (www.netbeans.org) se puede descargar la última versión.

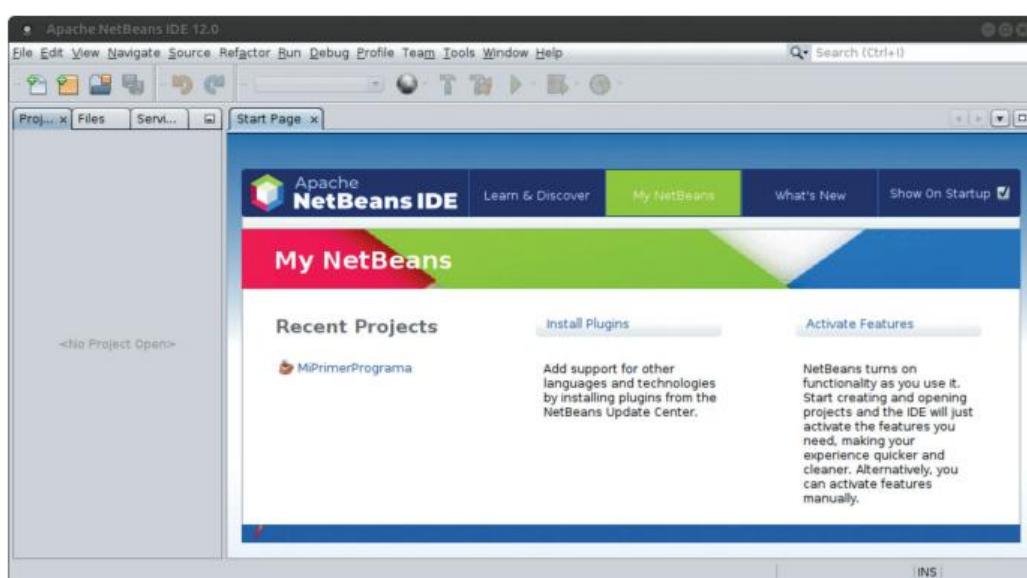


Figura 1.4. Ventana principal del IDE NetBeans.

Existen otros entornos de desarrollo, cada uno con sus características. Te invitamos a probar y experimentar hasta que encuentres el que mejor se adapte a tus necesidades, aunque no difieren mucho unos de otros.

En NetBeans crear un proyecto significa crear un nuevo programa en el que escribiremos las sentencias en Java que necesitemos. En el menú *File/New Project...* se accede a la ventana representada en la Figura 1.5, donde podemos elegir el tipo de proyecto. NetBeans se puede utilizar para escribir programas en distintos lenguajes de programación.

1. Seleccionaremos el lenguaje y la opción que nos interese, en nuestro caso, *Java with Ant*.

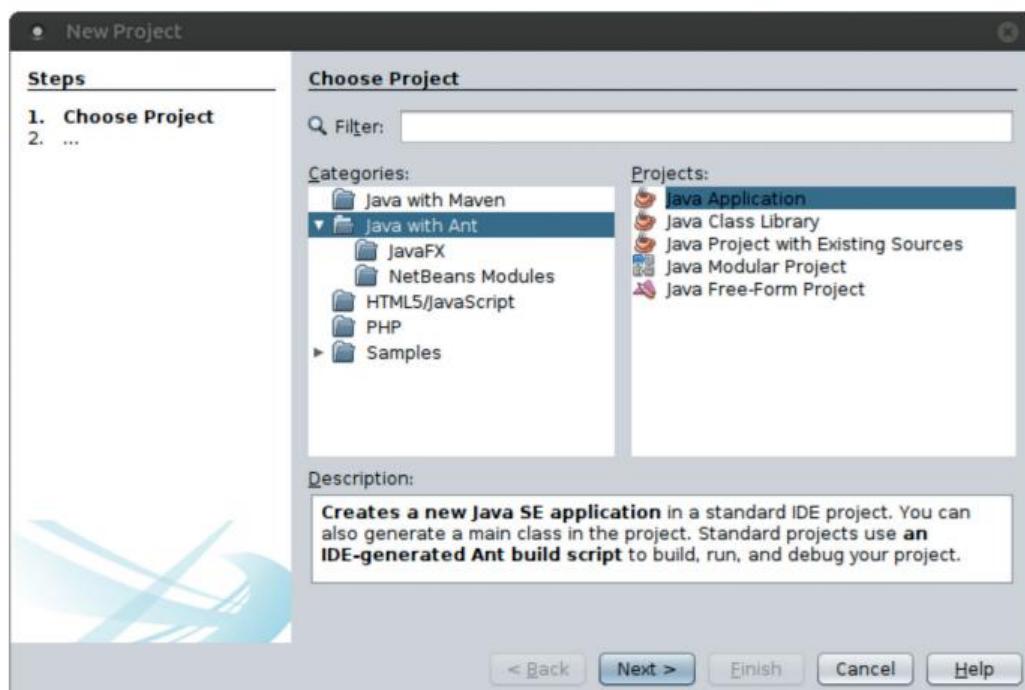


Figura 1.5. Selección del tipo de proyecto.

2. Elegiremos *Java Application*, donde el programa que diseñemos hará que el ordenador se comporte como una consola de entrada/salida, evitando elementos de programación avanzada como los gráficos de escritorio.

Pulsando en el botón *Next >* llegamos a una ventana (Figura 1.6), donde podremos elegir el nombre y la ubicación de nuestro proyecto. Hemos elegido como nombre *MiPrimerPrograma*. Marcaremos la casilla: *Create Main Class*.

El botón *Finish* finaliza el proceso; ahora se muestra un editor (Figura 1.7), donde podremos insertar las instrucciones que deseemos. NetBeans simplifica el trabajo creando el código del programa principal. Nosotros insertaremos las sentencias de nuestro programa entre las llaves de la función `main`, en la parte que aparece sombreada.

Una vez que hemos escrito las sentencias que necesitamos, para que comiencen a ejecutarse pulsaremos en el botón *Run Project*, representado por un triángulo verde que simula una tecla *Play* (véase Figura 1.8).

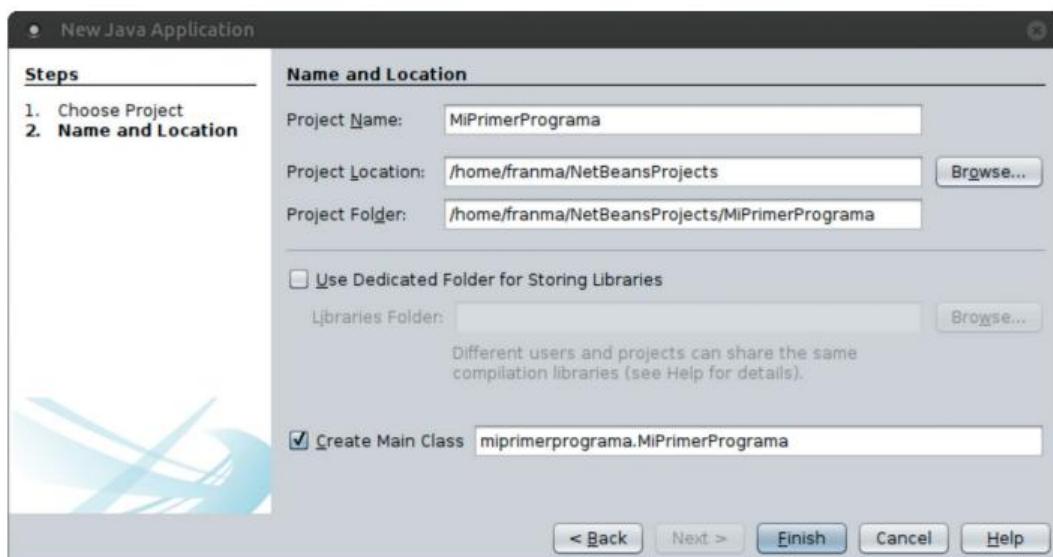


Figura 1.6. Nombre y localización del proyecto. NetBeans crea el nombre de la clase principal con el mismo nombre que el proyecto. En los ejercicios resueltos hemos modificado la clase principal para que se denomine Main.

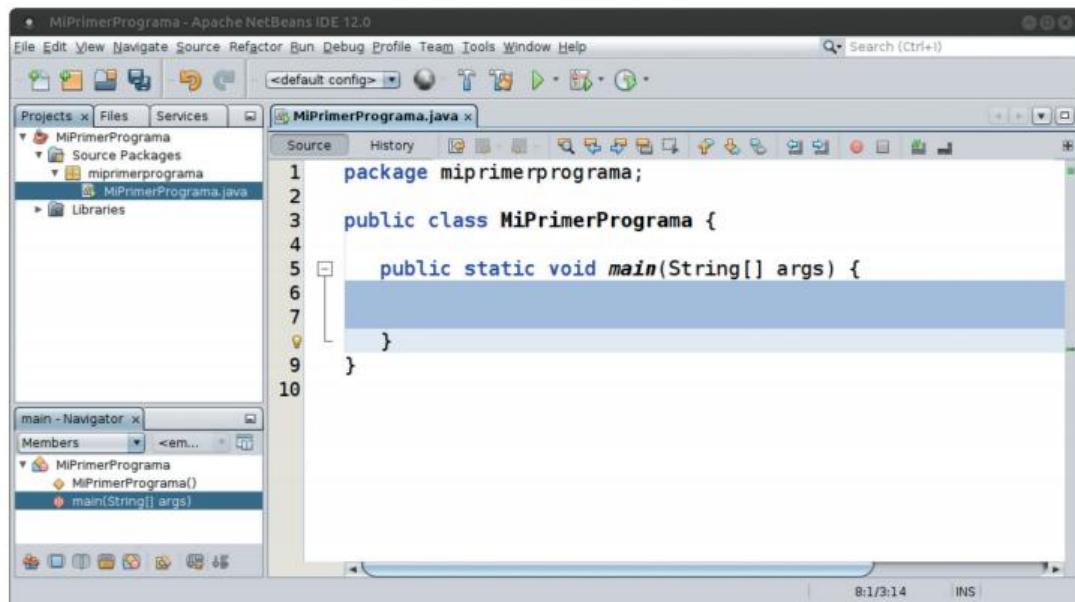


Figura 1.7. Editor de NetBeans. De momento, todo el código que escribamos irá siempre dentro de las llaves que determinan a la función main, es decir, nuestro código se escribirá en la zona sombreada.

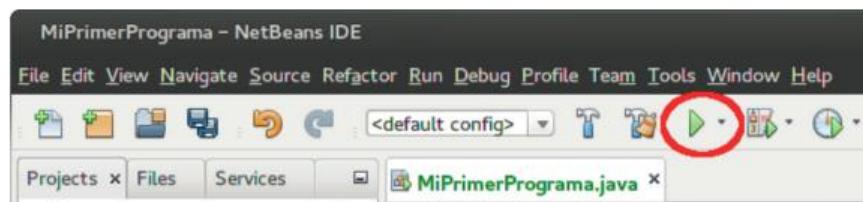


Figura 1.8. El botón rodeado en rojo es el botón Run Project, que permite ejecutar el código de nuestro proyecto. Otra alternativa es pulsar la tecla F6, que produce el mismo efecto.

■ 1.5. 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 MiPrimerPrograma {  
  
    public static void main(String[] args) {  
        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 NetBeans 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,

```
package miprimerprograma;
```

especifica que nuestro programa se agrupará en el paquete **miprimerprograma**. El nombre del paquete dependerá del nombre del proyecto; dicho de otra forma, NetBeans 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**.

Nota técnica



NetBeans asigna por defecto el nombre del proyecto como nombre de la clase principal y como nombre del paquete. Estos se pueden renombrar, tras seleccionarlos, mediante el menú *Refactor/Rename*. Por homogeneidad, hemos decidido que el nombre de la clase principal sea **Main**. En los ficheros con la solución de los ejercicios que se pueden descargar, previo registro, de la web de la Editorial Paraninfo (www.paraninfo.es), se ha omitido la sentencia **package**. Deberás decidir tú cómo se llamarán los proyectos y paquetes.

■ 1.6. 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 1.9, 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 1.9. Palabras reservadas de 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.

■ 1.7. 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 asignación 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.

■ ■ 1.7.1. 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`.

1.8. 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:

nota

1	0
---	---

Nota técnica



El tamaño de la memoria en un ordenador se mide en bytes —ocho bits (cero o uno)— y no en dígitos, como lo estamos haciendo aquí. Para comprender el concepto de tipo supondremos, por ahora, que el tamaño de la memoria se mide en dígitos.

Por el contrario, si deseamos utilizar calificaciones comprendidas entre 0 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):

nota

1	2	5
---	---	---

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 (Unidad 7) veremos cómo crear otros.

■■■ 1.8.1. 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.

Nota técnica



El tipo primitivo `char` está pensado para almacenar un solo carácter. El tipo primitivo `boolean` está diseñado para guardar tan solo dos posibles valores, que pueden representar: sí o no, cierto o falso; u otros dos conceptos antagónicos cualesquiera, como el día y la noche. Los dos posibles valores que puede almacenar una variable booleana se determinan por los literales: `true` y `false`.

■■■ 1.8.2. Rangos

En la Tabla 1.1 se describe el tamaño —espacio que ocupa en memoria— y el rango de valores que puede almacenar cada tipo primitivo.

Tabla 1.1. Tipos primitivos en Java. Puede apreciarse que cuanto mayor es el espacio que ocupa, mayor es el rango de valores que puede albergar

Tipo	Uso	Tamaño	Rango
byte	entero corto	8 bits	de -128 a 127
short	entero	16 bits	de -32 768 a 32 767
int	entero	32 bits	de -2 147 483 648 a 2 147 483 647
long	entero largo	64 bits	$\pm 9 223 372 036 854 775 808$
float	real precisión sencilla	32 bits	de -10^{32} a 10^{32}
double	real precisión doble	64 bits	de -10^{300} a 10^{300}
boolean	lógico	1 bit	true o false
char	texto	16 bits	cualquier carácter

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 sobreescrito. 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.

Por una parte, Java impide que asignemos a una variable un valor fuera del rango permitido por el tipo al que pertenece,

```
byte a = 300;
```

que produce un error del compilador, ya que el tipo byte posee un rango cuyos valores están comprendidos entre -128 y 127.

Por otra parte, para evitar el desbordamiento como resultado de un cálculo, los rangos en Java funcionan de forma circular; cuando se sobrepasa el valor máximo, se vuelve al valor mínimo del rango, y viceversa. Para el caso de una variable de tipo byte, la forma de contar sería:

0, 1, 2, ..., 126, 127, -128, -127, ..., -2, -1, 0, 1, ...

Expresado de otra forma, el valor máximo de un tipo más 1 no es un valor fuera de rango, sino el valor mínimo permitido para ese tipo. Y ocurre lo mismo con el valor mínimo. Por ejemplo, después de las sentencias,

```
byte a = 127;
a = a + 1;
```

Recordemos que el valor máximo para el tipo byte es 127, por lo tanto la variable a tendrá el valor -128, que es el siguiente a 127.

En el Apartado 1.12.2 veremos a fondo el funcionamiento de `System.out.println()`, pero nos adelantamos para que puedas visualizar el valor de las variables usadas en los ejemplos. La forma de mostrar en pantalla el valor de la variable `a` es escribiendo,

```
System.out.println(a);
```

■ 1.9. Variables de objeto

Es posible declarar variables cuyo tipo no sea un tipo primitivo, sino una clase. Por ahora, pensaremos en las clases como tipos de datos complejos, hasta que las estudiemos en profundidad en la Unidad 7.

A estas variables se les denomina *variables de objeto* y las utilizaremos para poder aprovechar las herramientas que Java proporciona.

De igual manera que las variables de tipos primitivos se inicializan por defecto, si en su declaración no se les asigna un valor, las variables de objeto se inicializan por defecto con el valor especial `null`, que representa que la variable se encuentra vacía.

En la regla de estilo que sigue Java, el identificador de las variables comienzan por minúscula, mientras que los nombres de las clases comienza por mayúscula. Esto permite, de un vistazo, distinguir en el código qué es cada uno de los identificadores usados.

■ 1.10. 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;
```

Recuerda



La línea anterior describe la sintaxis del lenguaje, es decir, el orden en el que se escriben las cosas. Para declarar una constante hemos de sustituir la palabra `tipo` por cualquiera de los tipos primitivos de Java: `int`, `char`, `byte`, etcétera.

Igualmente, `nombreConstante` puede sustituirse por cualquier nombre que nos guste para la constante. Por ejemplo: `PI`, `NUMERO_MAXIMO`, `IVA`, etcétera.

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 byte 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.

■ 1.11. 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 comentario multilínea, con la diferencia de que, para iniciararlo, 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; //declaramos la variable numeroPaginas como un entero  
/** este comentario será utilizado en caso de utilizar una herramienta  
de generación automática de documentación */
```

■ 1.12. API de Java

Una de las grandes ventajas de los lenguajes de programación modernos es que disponen de una amplia biblioteca de herramientas que realizan tareas complejas de forma transparente al programador que las utiliza, facilitando su tarea.

En el caso de que un lenguaje de programación no disponga de alguna herramienta específica, es necesario que sea el propio programador quien la construya, con los inconvenientes que esto conlleva. Es indudable que el hecho de disponer de herramientas facilita la labor de programar: por un lado se ahorra tiempo y trabajo, al no tener que implementarlas; y por otro, aporta un extra de seguridad al tener la certeza de que estas herramientas funcionan bien, ya que han sido diseñadas y comprobadas por programadores expertos.

Ilustraremos esta idea con un ejemplo de la vida cotidiana: supongamos que deseamos colgar un bonito cuadro en el salón de nuestra casa. Para ello es primordial hacer un taladro en la pared para colocar algún tipo de gancho o alcayata. Si no disponemos de un taladro eléctrico, tendremos que construirlo, lo que nos obliga a tener conocimientos de mecánica y electricidad, entre otras cosas, a la vez que es una tarea que ocupa nuestro tiempo. Es más, una vez construido tendremos dudas sobre su calidad. Si no lo hemos hecho bien, podría fallar al usarse. Es mucho más cómodo disponer de antemano del taladro eléctrico. Las herramientas que acompañan a un lenguaje de programación —al igual que el taladro eléctrico— están a nuestra disposición para facilitar las tareas cotidianas y liberarnos del trabajo de construirlas.

A estas herramientas, en Java, se les denomina **clases** y facilitan multitud de tareas. Algunos ejemplos de las funcionalidades que nos brindan son:

- **Lectura de datos:** leen información desde el teclado, desde un fichero o desde otros dispositivos.
- **Cálculos complejos:** realizan operaciones matemáticas como raíces cuadradas, logaritmos, cálculos trigonométricos, etcétera.
- **Manejo de errores:** controlan la situación cuando se produce un error de algún tipo.
- **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».

■ ■ 1.12.1. 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`.

A su vez, una clase proporciona una o varias funcionalidades, que se denominan métodos. Si consideramos el taladro eléctrico como una clase, nos proporciona dos funcionalidades —dos métodos—: taladrar y atornillar. `Math`, entre otros, dispone de los métodos `sqrt()`, que calcula una raíz cuadrada, `abs()`, que calcula el valor absoluto de un número, o de `random()`, que selecciona un número aleatorio.

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.time.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`. La sentencia `import` se coloca justo debajo de la declaración del paquete, como en la Figura 1.12. Tras importar una clase, ya no es necesario escribir su nombre cualificado,

```
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 ocasiones tener que importar una a una las clases de un paquete puede ser algo tedioso. Es posible importar todos las clases de un paquete mediante un asterisco:

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

Nota técnica

Mientras escribimos código, NetBeans analiza lo que escribimos y comprueba si usamos alguna clase que aún no ha sido importada. En este caso, mediante un triángulo amarillo junto a la línea de código, nos permite importar la clase con un solo clic.



Hay que tener cuidado al seleccionar con el ratón qué clases deseamos importar, ya que en ocasiones existen varias posibilidades. Existen distintas clases con el mismo nombre (aunque realizan funciones distintas y se ubican en paquete distintos).

Con respecto al mecanismo de importación, el paquete `java.lang` es una excepción. Al albergar clases fundamentales para Java —sin las cuales sería prácticamente imposible programar—, se necesitan sus clases en casi cualquier programa. Por este motivo, es el propio compilador el que importa de forma automática todas las clases del paquete `java.lang`, sin que tengamos que hacerlo nosotros. Es decir, podemos utilizar cualquier clase del paquete `java.lang` mediante su nombre corto sin tener que preocuparnos de su importación.

Por otra parte, cada clase que compone la API puede utilizarse de dos modos:

- **De forma estática:** se usa directamente el método. Por ejemplo, la clase `Math` se utiliza de forma estática. Veamos como calcular la raíz cuadrada de 16:

```
raiz = Math.sqrt(16); //calcula la raíz cuadrada de 16, que resulta 4.0
```

- **De forma no estática:** esta manera de utilizar las clases requiere del operador `new`, que se verá en profundidad en unidades posteriores. Un ejemplo de clase que se utiliza de esta forma es `Scanner`, que permite que el usuario introduzca datos en una aplicación. Por ahora, utilizaremos la clase `Scanner` como una fórmula literal (véase el Apartado 1.12.3).

Hasta aquí hemos visto las clases como un conjunto de herramientas, pero en la Unidad 7 las estudiaremos a fondo, veremos cómo implementar nuestras propias clases y todo lo relacionado con ellas. Por ahora, nos limitaremos a utilizar algunas clases sin más.

1.12.2. Salida por consola

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).

Un ejemplo de cómo `System.out.print()` muestra la salida por consola se aprecia en la ventana de Output (Figura 1.10).

Nota técnica

Para hacer visible la ventana de Output, pulsa *Output* en el menú *Window* de la barra de menú.



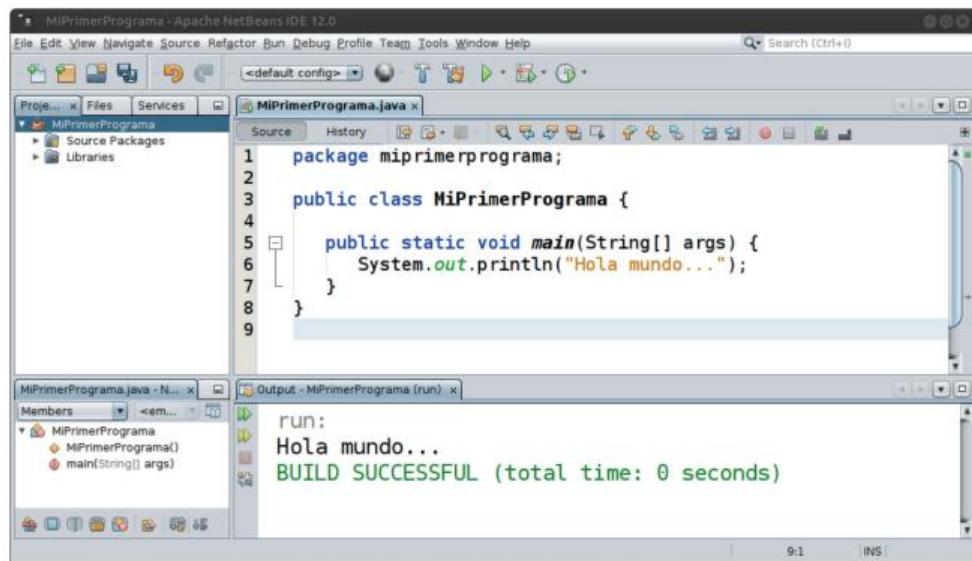


Figura 1.10. Salida por consola de la ejecución del programa, que mostrará el mensaje: «Hola mundo...».

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 —NetBeans— tiene un truco para escribir `System.out.println`: basta con escribir `sout` y pulsar la tecla tabulador. NetBeans 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.

Hay que tener en cuenta que `System.out.print` no inserta ningún retorno de carro al final del mensaje. Un ejemplo de ello (véase la variable `queHoraEs` en el Apartado 1.12.1):

```
System.out.println("Hola."); //inserta un retorno de carro
System.out.print("La hora del sistema es:"); //sin retorno de carro
System.out.print(queHoraEs); //muestra la hora hasta los milisegundos
```

El resultado será:

Hola.
La hora del sistema es:10:20:32.294

Los dos últimos mensajes se encuentran unidos, ya que no existe ningún retorno de carro después de «La hora del sistema es:». Donde queramos insertarlo, usaremos:

```
System.out.println()
```

o el carácter especial `\n`, que equivale a un retorno de carro.

```
System.out.println("Hola."); //inserta un retorno de carro  
System.out.print("La hora\n del sistema es:\n"); //dos retornos de carro  
System.out.print(queHoraEs);
```

Aparece en pantalla:

```
Hola.  
La hora  
del sistema es:  
10:20:32.294
```

Actividad resuelta 1.1

Escribir unas líneas de código que saluden al usuario con el mensaje: «Hola. Encantado de conocerlo.».

Solución

```
public class Main {  
    public static void main(String[] args) {  
        System.out.print("Hola. Encantado de conocerlo."); //salida por consola  
    }  
}
```

■ ■ ■ 1.12.3. Entrada de datos

Otra operación muy utilizada, disponible en la API —mediante la clase `Scanner`—, consiste en recabar información del usuario a través del teclado. Cuando se hace de forma simple, 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:

- `sc.nextInt()`: lee un número entero (`int`) por teclado.
- `sc.nextDouble()`: lee un número real (`double`).
- `sc.nextLine()`: lee una cadena de caracteres (una frase) hasta que se pulsa Intro.
- `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 1.11):

1. Se detiene la ejecución del programa y se espera a que el usuario teclee.

Recuerda



Para escribir, seleccionar previamente la ventana de *Output*.

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.

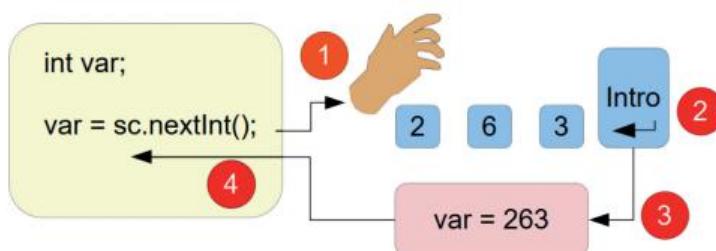


Figura 1.11. Entrada por consola. La clase `Scanner` interpreta la información tecleada por el usuario y la convierte en valores que son asignados a las variables.

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, a través de la variable numero
System.out.println("Ha escrito: " + numero);
```

Recuerda



Antes de escribir el número o cualquier otra cosa que solicitemos con un `Scanner`, tendremos que hacer clic con el ratón en la zona de *Output*. De lo contrario, lo que escribamos se insertará en la zona de código.

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: nextInt(), 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;
```

Mientras `java.lang` contiene clases fundamentales a la hora de programar, es decir, es prácticamente imposible escribir un programa sin utilizar ninguna de ellas; el paquete `java.util` contiene clases muy útiles, pero no imprescindibles.

Una vez creado `sc` podemos utilizarlo para leer tantas veces como necesitemos:

```
Scanner sc = new Scanner(System.in);
edad = sc.nextInt(); //leemos un entero
precio = sc.nextDouble(); //leemos un real
```

Hay que tener en cuenta que, al utilizar `sc.nextDouble()`, que lee un número real por teclado, tenemos que introducir los números con coma decimal (,) —por ejemplo: 12,3— en lugar de punto (.) —12.3—. Si queremos introducir los decimales con punto, debemos añadir la línea:

```
sc.useLocale(Locale.US);
```

justo después de crear el escáner. Para ello, antes debemos importar la clase `Locale`:

```
import java.util.Locale;
```

Para escribir nuestro programa utilizaremos siempre la estructura que se representa en la Figura 1.12. Finalmente, lo único que nos queda es seguir aprendiendo a programar con Java. Esto requiere practicar mucho sin dejar de disfrutar.

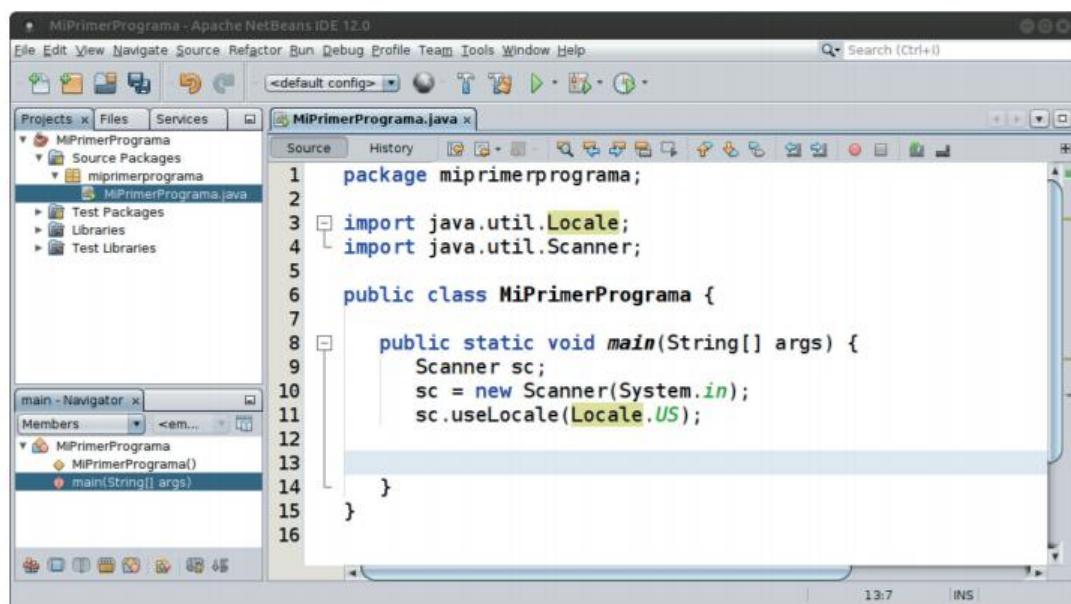


Figura 1.12. Programa principal donde se usa `Locale`; con esto podemos introducir los números decimales con punto, en lugar de coma, como indicador de la parte decimal. También se aprecia que se han usado varios `import`.

Actividad resuelta 1.2

Diseñar un programa que pida un número al usuario —por teclado— y a continuación lo muestre.

Solución

```
import java.util.Scanner;
/* En este ejercicio se pide un número y después se muestra tal cual. En este caso no
 * procesamos el dato de entrada. Esto no es un caso típico, pero nos sirve para ir
 * mostrando las distintas herramientas de las que disponemos. */
```

```

public class Main {
    public static void main(String[] args) {
        int num; //en la variable num guardaremos el número que se introduzca
        System.out.print("Escriba un número: "); //salida por consola: mensaje
        Scanner sc = new Scanner(System.in);
        num = sc.nextInt(); // entrada por consola
        //ahora mostraremos el valor de la variable num
        System.out.println("Valor introducido: " + num); //salida: mensaje +
        //variable. Utilizando + podemos concatenar en la salida por consola
        //tantos mensajes y variables como necesitemos
    }
}

```

■ 1.13. Operaciones básicas

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

■ ■ 1.13.1. 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

```

En estas asignaciones la expresión asignada es un valor explícito y no una expresión que necesite ser evaluada. Estos valores explícitos se llaman *literales*. Por ejemplo, 123 es un literal entero, mientras que 2.5 es un literal `double`. En cambio,

```

total = 2 * a; //total toma como valor el resultado de:
//2 por el valor de la variable a (que es 3). Es decir 6.
total = total - 5; //a total se le asigna el valor de:
//total (que es 6) menos 5. Es decir 1.

```

Ahora a `total` se le asigna dos expresiones que necesitan ser evaluadas antes de la asignación. La primera es una multiplicación —operador `*`— y la segunda es una resta.

Desde que se declara una variable hasta que se le asigna el primer valor, ¿cuánto vale la variable? Java asigna provisionalmente valores por defecto a las variables, pero impide realizar operaciones con ellas hasta que realicemos la primera asignación. En otros len-

guajes de programación hay que tener mucho cuidado, ya que la política de variables sin asignar cambia. Por ejemplo, el lenguaje C asigna a la variable un valor al azar.

■■■ 1.13.2. 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 = -a; // 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.

Tabla 1.2. Operadores aritméticos

Símbolo	Descripción
<code>+</code>	Suma
<code>+</code>	Más unario: positivo
<code>-</code>	Resta
<code>-</code>	Menor unario: negativo
<code>*</code>	Multiplicación
<code>/</code>	División
<code>%</code>	Módulo
<code>++</code>	Incremento en 1
<code>--</code>	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;
```

En un programa el incremento o decremento de una variable es algo tan usual que estos operadores están pensados con el único propósito de ahorrar trabajo al programador a la hora de teclear, y de paso, hacer el código más compacto, legible y eficiente. Ambos operadores pueden utilizarse de forma prefija (`++a;`) o posfija (`a++;`), y su comportamiento es distinto. Cuando se utiliza como prefijo, el operador tiene precedencia sobre el resto de las operaciones de la expresión. Y usado como posfijo, se realiza antes cualquier otra operación, dejando el incremento para el final.

```

int a, b, c; //declaramos las variables de tipo entero
a = 1; //a la variable "a" le asignamos 1
b = a++; //primero asignamos el valor de "a" a "b", y después incrementamos "a"
c = ++a; //primero incrementamos "a", y después asignamos su valor a "c"

```

Después de estas líneas de código, a vale 3, b vale 1 y c vale 3.

En la segunda asignación, lo primero que se hace es copiar el valor actual de a y, a continuación, se incrementa. El orden de las operaciones es asignar (=) e incrementar (++) . En la tercera asignación el valor final de c es 3, debido a que la primera operación que se hace es incrementar a, y después asignamos el valor incrementado a la variable c.

Cuando un ordenador realiza operaciones aritméticas con decimales, la precisión que puede utilizar es limitada, lo que genera pequeños errores de cálculo.

Veamos un ejemplo:

```

double a = 1.0/10.0 + 2.0/10.0; //el resultado debería ser 3/10
a = a * 10.0; //el resultado debería ser 3
System.out.println(a); //muestra 3.0000000000000004

```

Esto es algo que hay que tener en cuenta, ya que el resultado de un programa puede ser distinto al esperado debido a los errores producidos por los decimales.

Actividad resuelta 1.3

Pedir al usuario su edad y mostrar la que tendrá el próximo año.

Solución

```

import java.util.Scanner;
/* En el ejercicio realizamos las tres fases típicas de cualquier aplicación:
 * - Entrada de datos: pedimos la edad
 * - Procesado: en este caso incrementar la edad en 1
 * - Salida de datos: mostrar los resultados */
public class Main {
    public static void main(String[] args) {
        int edad; //aquí guardaremos la edad del usuario
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba su edad: ");
        edad = sc.nextInt();
        edad = edad + 1; //el año que viene tendrá un año más
        //la línea anterior puede sustuirse por: edad++;
        //ahora mostraremos el valor de la variable edad
        System.out.println("El próximo año su edad será: " + edad + " años");
    }
}

```

Actividad resuelta 1.4

Escribir una aplicación que pida el año actual y el de nacimiento del usuario. Debe calcular su edad, suponiendo que en el año en curso el usuario ya ha cumplido años.

Solución

```

import java.util.Scanner;
/* La edad puede calcularse como la diferencia entre el año actual y el de
 * nacimiento. Esto puede contener un error, en el caso de que en la fecha
 * actual aun no se haya celebrado el cumpleaños del año en curso.
 * Supondremos que el cumpleaños del usuario ya ha tenido lugar este año. */
public class Main {
    public static void main(String[] args) {
        int aActual; //año en curso (actual)
        int aNacimiento; //año de nacimiento
        int edad;
        Scanner sc = new Scanner(System.in);
        //leemos los datos
        System.out.print("Año de nacimiento: ");
        aNacimiento = sc.nextInt();
        System.out.print("Año actual: ");
        aActual = sc.nextInt();
        edad = aActual - aNacimiento; //calculamos la edad
        System.out.println("Su edad es: " + edad + " años.");
    }
}

```

Actividad resuelta 1.5

El tipo `short` permite almacenar valores comprendidos entre -32 768 y 32 767. Escribir un programa que compruebe que el rango de valores de un tipo se comporta de forma cíclica, es decir, el valor siguiente al máximo es el valor mínimo.

Solución

```

// Veremos como Java evita que una operación provoque un desbordamiento.
public class Main {
    public static void main(String[] args) {
        short num;
        num = 32767; //valor máximo dentro del rango de short
        System.out.println("Valor máximo para el tipo short: " + num);
        num++; //incrementamos en 1. Para evitar salirse del rango, la
        //variable num tomará el valor mínimo para el tipo short
        System.out.println("Valor mínimo para el tipo short: " + num);
    }
}

```

Actividad resuelta 1.6

Crear una aplicación que calcule la media aritmética de dos notas enteras. Hay que tener en cuenta que la media puede contener decimales.

Solución

```

import java.util.Scanner;
/* Pediremos dos notas enteras y calcularemos la media. Como la media puede
 * tener decimales utilizaremos una variable de tipo real. */

```

```

public class Main {
    public static void main(String[] args) {
        int nota1, nota2; //variables enteras para las notas
        double media; //la media puede contener decimales: usamos double
        Scanner sc = new Scanner(System.in);
        //leemos las notas
        System.out.print("Nota 1: ");
        nota1 = sc.nextInt();
        System.out.print("Nota 2: ");
        nota2 = sc.nextInt();
        //calculamos la media
        media = (nota1 + nota2) / 2.0;
        //en la expresión, el punto decimal de 2.0 hace que no sea una división entera.
        //El numerador sufre una conversión automática a real en doble precisión y el
        //resultado conserva la parte decimal
        System.out.println("La media es: " + media);
    }
}

```

Actividad resuelta 1.7

Diseñar una aplicación que calcule la longitud y el área de una circunferencia. Para ello, el usuario debe introducir el radio (que puede contener decimales).

Recordamos:

$$\text{Longitud} = 2\pi \cdot \text{radio}$$

$$\text{Área} = \pi \cdot \text{radio}^2$$

Solución

```

import java.util.*;
/* Para calcular la longitud y el área utilizaremos el valor de pi que nos brinda Math.
 * Y usaremos el método de la API que eleva una base a un exponente para el cuadrado. */
public class Main {
    public static void main(String[] args) {
        double radio; //el radio puede contener decimales
        double area, longitud;
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //usaremos decimales con .
        // pedimos los datos
        System.out.print("Escriba el radio: ");
        radio = sc.nextDouble();
        longitud = 2 * Math.PI * radio; //la clase Math pertenece al paquete
        //java.lang, que se importa por defecto
        area = Math.PI * Math.pow(radio, 2); //Math.pow(base, exponente) eleva la base
        //al exponente utilizado. Math.pow(radio, 2) eleva el radio a 2 (al cuadrado)
        System.out.println("La longitud de la circunferencia es: " + longitud);
        System.out.println("El área del círculo es: " + area);
    }
}

```

■ ■ ■ 1.13.3. 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:

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

Tabla 1.3. Operadores relacionales.

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

Como se vio en el apartado anterior, las operaciones aritméticas con decimales llevan implícitos pequeños errores de cálculo. Esto es algo que tener en cuenta cuando se realizan comparaciones. Sea:

```
double a = (1.0/10.0 + 2.0/10.0) * 10; //a debería valer 3,  
//pero vale 3.0000000000000004
```

Si realizamos la comparación `a == 3`, resultará falso. En lugar de comparar valores de tipo `float` o `double` directamente, lo que se hace es realizar una resta y si el resultado está por debajo de un umbral, se asume que los valores son iguales. Es decir, si `a - 3 <= 0.0000001` es `true`, se asume que `a` es igual a 3.

Actividad resuelta 1.8

Realizar una aplicación que solicite al usuario su edad y le indique si es mayor de edad (mediante un literal booleano: `true` o `false`).

Solución

```
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba su edad: ");
        int edad = sc.nextInt();
        boolean mayorEdad = edad >= 18; //ser mayor de edad implica que la
                                         //edad sea mayor o igual que 18
        System.out.println("Mayoria de edad: " + mayorEdad);
    }
}
```

Actividad resuelta 1.9

Escribir un programa que pida un número al usuario e indique mediante un literal booleano (`true` o `false`) si el número es par.

Solución

```
import java.util.*;
/* Los números pares tienen la propiedad que al dividirlos por dos la división es exacta,
 * es decir, el resto (módulo) de la división siempre es 0. */
public class Main {
    public static void main(String[] args) {
        int numero;
        System.out.print("Escriba un número: ");
        //Es habitual crear y leer de un Scanner, haciendo todo en la misma instrucción
        numero = new Scanner(System.in).nextInt();
        boolean esPar = (numero % 2) == 0; //calcula el resto de dividir el número
                                         //entre 2 y el resultado de esta operación la compara con 0
        System.out.println("Es par: " + esPar);
    }
}
```

1.13.4. 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* (conjunction Y), *or* (disyunción O) y *not* (negación).

Tabla 1.4. Operadores lógicos en Java

Símbolo	Descripción
<code>&&</code>	Operador and: Y
<code> </code>	Operador or: O
<code>!</code>	Operador not: negación

La expresión formada a partir de otras dos unidas por el operador `and` será `true` cuando ambas expresiones utilizadas se evalúen como ciertas. Y en caso contrario, es decir, si alguna o las dos expresiones se evalúan como falsas, la expresión resultante también será falsa.

```
exprA && exprB
```

La expresión anterior será cierta solo cuando `exprA` y `exprB` sean ciertas. Veamos las siguientes expresiones:

- `3 <= 5 && 2 == 2`. ¿Es 3 menor o igual que 5 y a la vez, es 2 igual a 2? Cierto, ya que tanto la expresión `3 <= 5` como `2 == 2` son ciertas.
- `3 <= 5 && 2 > 10`. ¿Es 3 menor o igual que 5 y a la vez, es 2 mayor que 10? Falso, ya que al menos una expresión utilizada es falsa (`2 > 10`).

El operador `or` —`o`— será cierto cuando alguna de las expresiones que lo forman sea cierta. En caso contrario será falso.

```
exprA || exprB
```

La expresión se evaluará cierto cuando `exprA` sea cierta, cuando `exprB` sea cierta o cuando ambas sean ciertas.

- `1 != 2 || 5 < 3`. ¿Es cierto que 1 sea distinto de 2 o que 5 sea menor que 3? La primera expresión es cierta: 1 es distinto de 2, mientras que la segunda expresión es falsa. Por tanto, la expresión completa se evalúa como verdadera.

El operador `not` —negación— es un operador unario que cambia los valores booleanos de cierto a falso y viceversa.

- `!(1 < 2)`. La expresión se evalúa como la negación —lo contrario— de 1 menor que 2, que es verdadera. Por tanto, la expresión completa se evalúa falsa.

Actividad resuelta 1.10

Diseñar un algoritmo que nos indique si podemos salir a la calle. Existen aspectos que influirán en esta decisión: si está lloviendo y si hemos terminado nuestras tareas. Solo podremos salir a la calle si no está lloviendo y hemos finalizado nuestras tareas. Existe una opción en la que, indistintamente de lo anterior, podremos salir a la calle: el hecho de que tengamos que ir a la biblioteca (para realizar algún trabajo, entregar un libro, etc.). Solicitar al usuario (mediante un booleano) si llueve, si ha finalizado las tareas y si necesita ir a la biblioteca. El algoritmo debe mostrar mediante un booleano (`true` o `false`) si es posible que se le otorgue permiso para ir a la calle.

Solución

```
//Tras solicitar la información requerida usaremos operadores lógicos para conseguir
//saber si es posible salir a la calle.
public class Main {
    public static void main(String[] args) {
        boolean llueve, finalizadoTareas, irBiblioteca;
        Scanner sc = new Scanner(System.in);
        System.out.println("¿Está lloviendo? (true/false)");
        llueve = sc.nextBoolean();
        finalizadoTareas = sc.nextBoolean();
        irBiblioteca = sc.nextBoolean();
        boolean salida = false;
        if (!llueve && finalizadoTareas) {
            salida = true;
        } else if (irBiblioteca) {
            salida = true;
        }
        System.out.println("Puedes salir: " + salida);
    }
}
```

```

llueve = sc.nextBoolean();
System.out.println("¿Has finalizado tus tareas? (true/false)");
finalizadoTareas = sc.nextBoolean();
System.out.println("¿Tienes que salir a la biblioteca? (true/false)");
irBiblioteca = sc.nextBoolean();
boolean salir = !llueve && finalizadoTareas || irBiblioteca;
System.out.println("Puedes salir: " + salir);
}
}

```

1.13.5. 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.

Tabla 1.5. Operadores opera y asigna

Símbolo	Descripción
<code>+=</code>	Suma y asigna
<code>-=</code>	Resta y asigna
<code>*=</code>	Multiplica y asigna
<code>/=</code>	Divide y asigna
<code>%=</code>	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;
```

Actividad resuelta 1.11

Un frutero necesita calcular los beneficios anuales que obtiene de la venta de manzanas y peras. Por este motivo, es necesario diseñar una aplicación que solicite las ventas (en kilos) de cada semestre para cada fruta. La aplicación mostrará el importe total sabiendo que el precio del kilo de manzanas está fijado en 2,35 € y el kilo de peras en 1,95 €.

Solución

```

import java.util.*;
/* Los datos de entrada que necesitamos son:
 * - cantidad vendida en el semestre 1 (de peras y manzanas)
 * - cantidad vendida en el semestre 2 (ídem) */
public class Main {
    public static void main(String[] args) {
        final double PRECIO_MANZANAS = 2.35; //valores constantes, no varian durante
        //el programa
        final double PRECIO_PERAS = 1.95;
        //los identificadores de constantes los escribimos en mayúsculas
        int vManz1Sem, vManz2Sem; //ventas (en kilos) por semestre
        int vPeras1Sem, vPeras2Sem; //igual para las peras
        double impTotal; //importe total
        Scanner sc = new Scanner(System.in);
        //pedimos los datos
        System.out.println("Para las manzanas: ");
        System.out.print("Ventas (en kilos) del primer semestre: ");
        vManz1Sem = sc.nextInt();
        System.out.print("Ventas (en kilos) del segundo semestre: ");
        vManz2Sem = sc.nextInt();
        System.out.println("Para las peras: ");
        System.out.print("Ventas (en kilos) del primer semestre: ");
        vPeras1Sem = sc.nextInt();
        System.out.print("Ventas (en kilos) del segundo semestre: ");
        vPeras2Sem = sc.nextInt();
        //calculamos el importe total obtenido
        impTotal = (vManz1Sem + vManz2Sem) * PRECIO_MANZANAS;
        impTotal += (vPeras1Sem + vPeras2Sem) * PRECIO_PERAS;
        System.out.println("El importe total es de: " + impTotal + " euros");
    }
}

```

■ ■ ■ 1.13.6. 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:

expresiónCondicional ? 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 el valor 20

```

Actividad resuelta 1.12

Escribir un programa que pida un número al usuario y muestre su valor absoluto.

Solución

```
import java.util.*;
/* Dado un número, para calcular su valor absoluto solo tenemos que saber si es
 * negativo, en cuyo caso es necesario multiplicarlo por -1, con lo que se
 * consigue el mismo valor pero con signo positivo.*/
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba un número (positivo o negativo): ");
        int n = sc.nextInt();
        int valorAbsoluto = n < 0 ? -1 * n : n;
        System.out.println("El valor absoluto de " + n + " es " + valorAbsoluto);
    }
}
```

Otra solución consiste en utilizar el método `Math.abs()`, que calcula el valor absoluto de un número.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba un número (positivo o negativo): ");
        int n = sc.nextInt();
        int valorAbsoluto = Math.abs(n);
        System.out.println("El valor absoluto de " + n + " es " + valorAbsoluto);
    }
}
```

1.13.7. 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.

- 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).
- 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.

Tabla 1.6. Precedencia de operadores. Los operadores con mayor precedencia (son las primeras operaciones que se realizan) son los posfijos, y los de asignación son los de menor precedencia

Descripción	Operador
Posfijos	<code>expr++ expr--</code>
Unarios prefijos	<code>++expr --expr +expr -expr !expr</code>
Aritméticos	<code>* / %</code>
Aritméticos	<code>+ -</code>
Relacionales	<code>< <= > >=</code>
Comparación	<code>== !=</code>
AND lógico	<code>&&</code>
OR lógico	<code> </code>
Ternario	<code>? :</code>
Asignación	<code>= += -= *= /= %= &= ^=</code>

1.14. 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 guardarla. 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`.

Nota técnica



Caso especial de conversión son los valores de tipo `char` que, con un tamaño de 16 bits, pueden convertirse a su valor entero en el código UNICODE, como se verá más adelante. Esto permite asignarlos a una variable `int`:

```
int c = 'a'; //que equivale a int c = 97;
ya que 97 es el código asignado al carácter 'a'.
```

Muy distinto es que intentemos asignar un valor `double` a una variable `int`, que no tiene sitio suficiente para guardarla. Lo normal 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;
```

Actividad resuelta 1.13

Escribir un programa que solicite las notas del primer, segundo y tercer trimestre (notas enteras que se solicitarán al usuario). El programa debe mostrar la nota media del curso como se utiliza en el boletín de calificaciones (solo la parte entera) y como se usa en el expediente académico (con decimales).

Solución

```
import java.util.Scanner;
/* Pediremos tres notas enteras y calcularemos la media con y sin decimales. */
public class Main {
    public static void main(String[] args) {
        int nota1, nota2, nota3; //variables para las notas
        int mediaBoletin; //la media es de tipo entero
        double mediaExpediente; //la media usa decimales
        Scanner sc = new Scanner(System.in);
        //leemos las notas
        System.out.print("Nota primer trimestre: ");
        nota1 = sc.nextInt();
        System.out.print("Nota segundo trimestre: ");
        nota2 = sc.nextInt();
        System.out.print("Nota tercer trimestre: ");
        nota3 = sc.nextInt();
        //calculamos la media con decimales
        mediaExpediente = (nota1 + nota2 + nota3) / 3.0; //el 3.0 fuerza una
            //división real
        mediaBoletin = (int) mediaExpediente; //convertimos un valor double en un
            //valor int, truncando la parte decimal.
            //Por tanto, hay pérdida de información.
        System.out.println("Boletín de calificaciones: " + mediaBoletin);
        System.out.println("Expediente académico: " + mediaExpediente);
    }
}
```

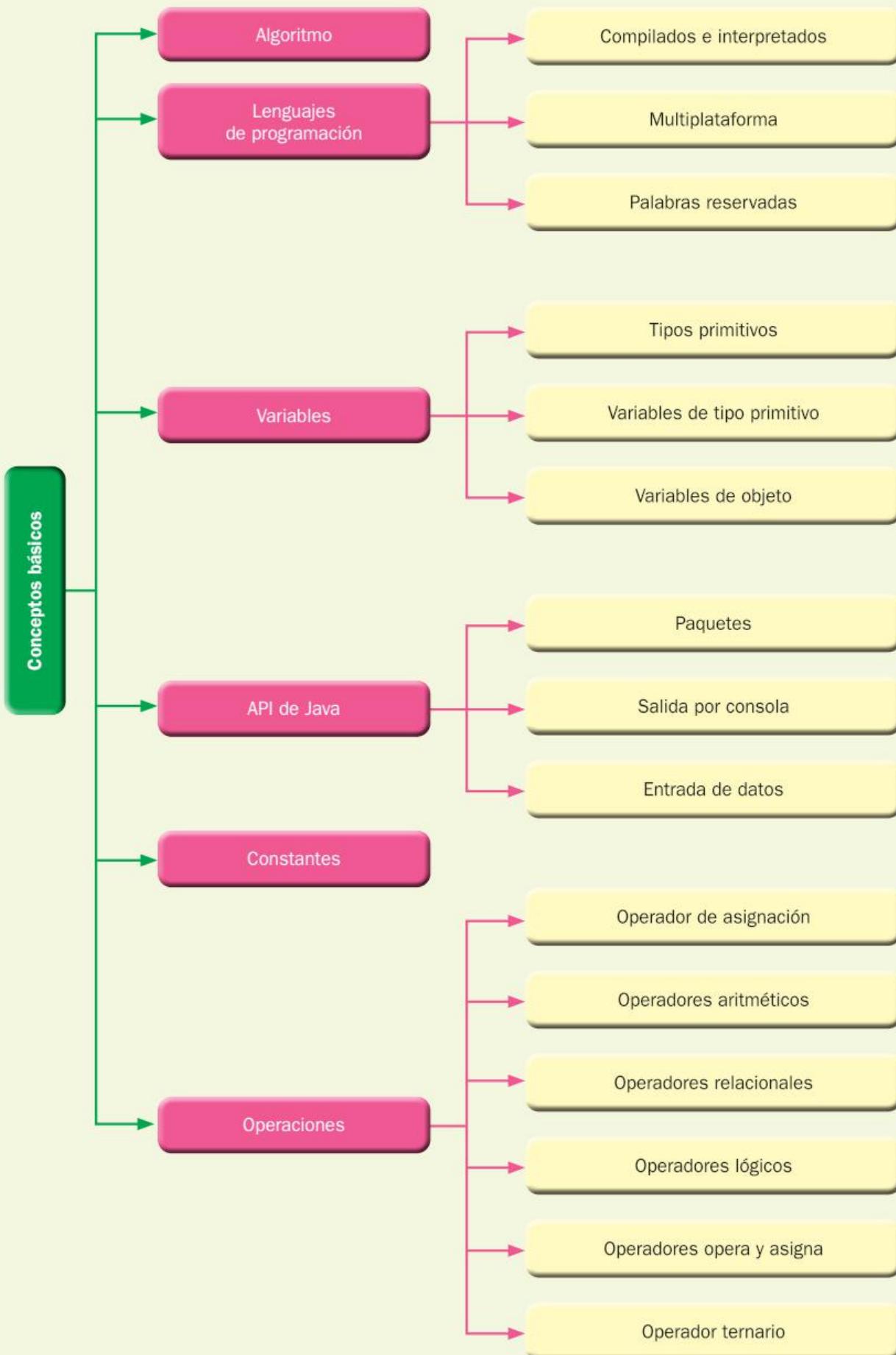
Actividad resuelta 1.14

Realizar un programa que pida como entrada un número decimal y lo muestre redondeado al entero más próximo.

Solución

```
import java.util.Locale; //importamos las dos clases que necesitamos
import java.util.Scanner;
//Cuando se importan múltiples clases de un mismo paquete, en lugar de escribir
//un import para cada clase, existe la opción de usar:
import java.util.*;//que importa las clases necesarias de java.util
/* Redondear un número decimal significa aproximararlo al entero más cercano.
 * Para ello, lo que haremos será sumar 0.5 y truncar (eliminar los decimales)
 * el resultado. Así los números:
 * 2.3 se redondea a 2
 * 4.8 se redondea a 5 */
```

```
public class Main {  
    public static void main(String[] args) {  
        double n; //aquí guardamos el número decimal introducido por el usuario  
        int redondeo; //utilizamos esta variable para truncar los decimales  
        Scanner sc = new Scanner(System.in);  
        sc.useLocale(Locale.US); //en lugar de coma utiliza punto para los decimales  
        System.out.print("Escriba un número decimal (con punto): ");  
        n = sc.nextDouble();  
        //ahora redondearemos n  
        redondeo = (int) (n + 0.5); //convertimos un real en un entero.  
        //Esta es una conversión por estrechamiento, por lo tanto estamos  
        //obligados a aplicar un cast (int). En caso de no hacerlo el  
        //compilador generará un error.  
        System.out.println(n + " redondeado es: " + redondeo);  
    }  
}
```



Actividades de comprobación

- 1.1.** ¿Cuál de los siguientes identificadores no puede emplearse para una variable?
 - a) language.
 - b) ultimo.
 - c) final.
 - d) fin.

- 1.2.** De todos los tipos primitivos disponibles en Java, selecciona cuál o cuáles son los que tienen un mayor tamaño y, por lo tanto, pueden albergar un mayor número de valores:
 - a) long.
 - b) long y double.
 - c) long, double y short.
 - d) En Java todos los tipos primitivos tienen el mismo tamaño.

- 1.3.** ¿Mediante qué símbolo es posible añadir un comentario en nuestro código?
 - a) #
 - b) //
 - c) <!--
 - d) Cualquiera de los anteriores.

- 1.4.** ¿Qué paquete se importa automáticamente en cualquier programa sin necesidad de tener que utilizar una sentencia `import`?
 - a) java.util
 - b) java.time
 - c) java.Scanner
 - d) java.lang

- 1.5.** ¿Cuáles de las siguientes instrucciones nos permiten mostrar información por consola?
 - a) new Scanner()
 - b) Math.sqrt()
 - c) System.out.println()
 - d) Message()

- 1.6.** ¿Qué instrucción es equivalente a: `i++`?
 - a) `i = i + 1`
 - b) `i = 1 + i`
 - c) `i += 1`
 - d) Cualquiera de los anteriores.

- 1.7.** Si evalúas la siguiente expresión: `2 < 1 || 2 != 1`, el resultado de dicha expresión es:
 - a) 1.
 - b) 2.
 - c) true.
 - d) false.

- 1.8.** ¿Qué valor toma la variable `a`, tras la ejecución de la instrucción: `int a = 1 < 2 ? 3 : 4;`?
- 1.
 - 2.
 - 3.
 - 4.
- 1.9.** Selecciona la expresión cuya evaluación resulta 3:
- $3 + 2 * 6 / 5$
 - $(3 + 2) * 6 / 5$
 - $(3 + 2 * 6) / 5$
 - $3 + 2 * (6 / 5)$
- 1.10.** En las siguientes conversiones de tipo, ¿cuál de ellas produce un error?
- `int a = (int) 1.23;`
 - `int a = 12.3;`
 - `double a = (double) 123;`
 - `double a = 123;`

Actividades de aplicación

- 1.11.** Un economista te ha encargado un programa para realizar cálculos con el IVA. La aplicación debe solicitar la base imponible y el IVA que se debe aplicar. Muestra en pantalla el importe correspondiente al IVA y al total.
- 1.12.** Escribe un programa que tome como entrada un número entero e indique qué cantidad hay que sumarle para que el resultado sea múltiplo de 7. Un ejemplo:
- A 2 hay que sumarle 5 para que el resultado ($2 + 5 = 7$) sea múltiplo de 7.
 - A 13 hay que sumarle 1 para que el resultado ($13 + 1 = 14$) sea múltiplo de 7.
- Si proporcionas el número 2 o el 13, la salida de la aplicación debe ser 5 o 1, respectivamente.
- Pista: El operador módulo puede ser muy útil para solucionar esta actividad.
- 1.13.** Modifica la Actividad de Aplicación 1.12 para que, indicando dos números n y m , diga qué cantidad hay que sumarle a n para que sea múltiplo de m .
- 1.14.** Crea un programa que pida la base y la altura de un triángulo y muestre su área.

$$\text{Área triángulo} = \frac{\text{base} \cdot \text{altura}}{2}$$

- 1.15.** Dado el siguiente polinomio de segundo grado:

$$y = ax^2 + bx + c$$

crea un programa que pida los coeficientes a , b y c , así como el valor de x , y calcula el valor correspondiente de y .

- 1.16.** Diseña una aplicación que solicite al usuario que introduzca una cantidad de segundos. La aplicación debe mostrar cuántas horas, minutos y segundos hay en el número de segundos introducidos por el usuario.
- 1.17.** Solicita al usuario tres distancias:
- La primera, medida en milímetros.
 - La segunda, medida en centímetros.
 - La última, medida en metros.
- Diseña un programa que muestre la suma de las tres longitudes introducidas (medida en centímetros).
- 1.18.** Un biólogo está realizando un estudio de distintas especies de invertebrados y necesita una aplicación que le ayude a contabilizar el número de patas que tienen en total todos los animales capturados durante una jornada de trabajo. Para ello, te ha solicitado que escribas una aplicación a la que hay que proporcionar:
- El número de hormigas capturadas (6 patas).
 - El número de arañas capturadas (8 patas).
 - El número de cochinillas capturadas (14 patas).
- La aplicación debe mostrar el número total de patas.
- 1.19.** Una empresa que gestiona un parque acuático te solicita una aplicación que les ayude a calcular el importe que hay que cobrar en la taquilla por la compra de una serie de entradas (cuyo número será introducido por el usuario). Existen dos tipos de entrada: infantiles, que cuestan 15,50 €; y de adultos, que cuestan 20 €.
- En el caso de que el importe total sea igual o superior a 100 €, se aplicará automáticamente un bono descuento del 5 %.
- 1.20.** Solicita al usuario un número real y calcula su raíz cuadrada. Implementa el programa utilizando el nombre cualificado de las clases, en lugar de utilizar ninguna importación.
- 1.21.** Pide dos números al usuario: *a* y *b*. Deberá mostrarse `true` si ambos números son iguales y `false` en caso contrario.
- 1.22.** La FILA (Federación Internacional de Lanzamiento de Algoritmo) realiza una competición donde cada participante escribe un algoritmo en un papel y lo lanza, ganando quien consiga lanzarlo más lejos. La peculiaridad del concurso es que la longitud del lanzamiento se mide en metros (con tantos decimales como se desee), pero para el ranking solo se tiene en cuenta la longitud en centímetros (sin decimales). Por ejemplo, para un lanzamiento de 12,3456 m (que son 1234,56 cm) solo se contabilizarán 1234 cm.
- Realiza un programa que solicite la longitud (en metros) de un lanzamiento y muestre la parte entera correspondiente en centímetros.

Actividades de ampliación

- 1.23. Busca en internet información sobre otros IDE utilizados habitualmente para desarrollar en Java. Realiza una comparativa con respecto a NetBeans.
- 1.24. Java es un lenguaje de programación que se basa en características de otros lenguajes que le han precedido. Busca las palabras reservadas del lenguaje C y, a partir de ellas, opina sobre qué características piensas que Java ha adquirido de C.
- 1.25. Existen muchos conceptos vinculados a los lenguajes de programación como, por ejemplo, los lenguajes compilados o interpretados. Investiga sobre qué son y cómo funcionan los lenguajes de programación:
 - Del lado del cliente en la web.
 - Del lado del servidor en la web.
 - Lenguajes de scripting.Realiza una puesta en común con el resto de la clase.
- 1.26. El tipo primitivo más usado para realizar cálculos es, sin duda, el `double`, ya que es el que tiene un mayor tamaño, lo que permite guardar números decimales con una mayor precisión. El problema que presenta este tipo es que, para cálculos cotidianos, su precisión es más que suficiente, pero para aplicaciones científico-técnicas, en ciertas ocasiones, su precisión puede quedarse algo corta.
Realiza una investigación en internet y encuentra una o varias clases, pertenecientes a la API de Java, que permitan realizar operaciones reales con una precisión tan alta como se necesite.
- 1.27. Además de los operadores que se han visto en esta unidad, Java dispone de otros que se utilizan para diferentes tipos de operaciones. Busca en la documentación oficial de Java algunos operadores que no hemos visto y realiza sencillos programas donde se prueba su utilización.
- 1.28. El índice TIOBE muestra una evolución mensual de los lenguajes de programación más usados. El número de proyectos que usa cada lenguaje se infiere a partir de las búsquedas que se realizan en distintos navegadores sobre ellos y a partir de lugares que albergan proyectos en distintas tecnologías.
Visita el índice TIOBE y comprueba en qué posición se encuentra Java y cuál ha sido su evolución en los últimos meses.