Plan general

2.1 Introducción

En este capítulo le presentaremos la programación de aplicaciones en Java. Empezaremos con ejemplos de programas que muestran (dan salida a) mensajes en la pantalla. Después veremos un programa que obtiene (da entrada a) dos números de un usuario, calcula la suma y muestra el resultado. Usted aprenderá cómo ordenar a la computadora que realice cálculos aritméticos y guardar sus resultados para usarlos más adelante. El último ejemplo demuestra cómo tomar decisiones. La aplicación compara dos números y después muestra mensajes con los resultados de la comparación. Usará las herramientas de la línea de comandos del JDK para compilar y ejecutar los programas de este capítulo. Si prefiere usar un entorno de desarrollo integrado (IDE), también publicamos videos Dive Into® en `http://www.deitel.com/books/jhtp10/` para Eclipse, NetBeans e IntelliJ IDEA.

2.2 Su primer programa en Java: impresión de una línea de texto

Una **aplicación** en Java es un programa de computadora que se ejecuta cuando usted utiliza el **comando `java`** para iniciar la Máquina Virtual de Java (JVM). Más adelante en esta sección hablaremos sobre cómo compilar y ejecutar una aplicación de Java. Primero vamos a considerar una aplicación simple que muestra una línea de texto. En la figura 2.1 se muestra el programa, seguido de un cuadro que muestra su salida.

```
1  // Fig. 2.1: Bienvenido1.java
2  // Programa para imprimir texto.
3
4  public class Bienvenido1
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
9        System.out.println("Bienvenido a la programacion en Java!");
10    } // fin del método main
11 } // fin de la clase Bienvenido1
```

Fig. 2.1 | Programa para imprimir texto (parte 1 de 2).

```
Bienvenido a la programacion en Java!
```

Fig. 2.1 | Programa para imprimir texto (parte 2 de 2). *Nota*: En los códigos se han omitido los acentos para evitar la incompatibilidad al momento de ejecutarse en distintos entornos.

El programa incluye números de línea, que incluimos para fines académicos; *no* son parte de un programa en Java. Este ejemplo ilustra varias características importantes de Java. Pronto veremos que la línea 9 se encarga del verdadero trabajo: mostrar la frase `Bienvenido a la programacion en Java!` en la pantalla.

Comentarios en sus programas

Insertamos **comentarios** para **documentar los programas** y mejorar su legibilidad. El compilador de Java *ignora* los comentarios, de manera que la computadora *no* hace nada cuando el programa se ejecuta.

Por convención, comenzamos cada uno de los programas con un comentario, el cual indica el número de figura y el nombre del archivo. El comentario en la línea 1

```
// Fig. 2.1: Bienvenido1.java
```

empieza con `//`, lo cual indica que es un **comentario de fin de línea**, el cual termina al final de la línea en la que aparecen los caracteres `//`. Un comentario de fin de línea no necesita empezar una línea; también puede estar en medio de ella y continuar hasta el final (como en las líneas 6, 10 y 11). La línea 2

```
// Programa para imprimir texto.
```

según nuestra convención, es un comentario que describe el propósito del programa.

Java también cuenta con **comentarios tradicionales**, que se pueden distribuir en varias líneas, como en

```
/* Éste es un comentario tradicional. Se puede
   dividir en varias líneas */
```

Estos comentarios comienzan y terminan con los delimitadores `/*` y `*/`. El compilador ignora todo el texto entre estos delimitadores. Java incorporó los comentarios tradicionales y los comentarios de fin de línea de los lenguajes de programación C y C++, respectivamente. Nosotros preferimos usar los comentarios con `//`.

Java también cuenta con un tercer tipo de comentarios: los **comentarios Javadoc**, que están delimitados por `/**` y `*/`. El compilador ignora todo el texto entre los delimitadores. Estos comentarios nos permiten incrustar la documentación de manera directa en nuestros programas. Dichos comentarios son el formato preferido en la industria. El **programa de utilería `javadoc`** (parte del JDK) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. En el apéndice G en línea, Creación de documentación con `javadoc`, demostramos el uso de los comentarios Javadoc y la herramienta `javadoc`.

Error común de programación 2.1

*Olvidar uno de los delimitadores de un comentario tradicional o Javadoc es un **error de sintaxis**, el cual ocurre cuando el compilador encuentra un código que viola las reglas del lenguaje Java (es decir, su sintaxis). Estas reglas son similares a las reglas gramaticales de un lenguaje natural que especifican la estructura de sus oraciones. Los errores de sintaxis se conocen también como **errores del compilador**, **errores en tiempo de compilación** o **errores de compilación**, ya que el compilador los detecta durante la compilación del programa. Al encontrar un error de sintaxis, el compilador genera un mensaje de error. Debe eliminar todos los errores de compilación para que su programa se compile de manera correcta.*

Buena práctica de programación 2.1

Ciertas organizaciones requieren que todos los programas comiencen con un comentario que explique su propósito, el autor, la fecha y la hora de la última modificación del mismo.

Tip para prevenir errores 2.1

Cuando escriba nuevos programas o modifique alguno que ya exista, mantenga sus comentarios actualizados con el código. A menudo los programadores tendrán que realizar cambios en el código existente para corregir errores o mejorar capacidades. Al actualizar sus comentarios, ayuda a asegurar que éstos reflejen con precisión lo que el código hace. Esto facilitará la comprensión y edición de su programa en el futuro. Los programadores que usan o actualizan código con comentarios obsoletos podrían realizar suposiciones incorrectas sobre el código, lo cual podría conducir a errores o incluso infracciones de seguridad.

Uso de líneas en blanco

La línea 3 es una línea en blanco. Las líneas en blanco, los espacios y las tabulaciones facilitan la lectura de los programas. En conjunto se les conoce como **espacio en blanco**. El compilador ignora el espacio en blanco.

Buena práctica de programación 2.2

Utilice líneas en blanco y espacios para mejorar la legibilidad del programa.

Declaración de una clase

La línea 4

```
public class Bienvenido1
```

comienza una **declaración de clase** para la clase `Bienvenido1`. Todo programa en Java consiste al menos de una clase que usted (el programador) debe definir. La **palabra clave `class`** introduce una declaración de clase, la cual debe ir seguida de inmediato por el **nombre de la clase** (`Bienvenido1`). Las **palabras clave** (también conocidas como **palabras reservadas**) se reservan para uso exclusivo de Java y siempre se escriben en minúscula. En el apéndice C se muestra la lista completa de palabras clave de Java.

En los capítulos 2 al 7, todas las clases que definimos comienzan con la palabra clave `public`. Por ahora, sólo recuerde que debemos usar `public`. En el capítulo 8 aprenderá más sobre las clases `public` y las que no son `public`.

Nombre de archivo para una clase `public`

Una clase `public` *debe* colocarse en un archivo que tenga el nombre de la forma *NombreClase*`.java`. Por lo tanto, la clase `Bienvenido1` se almacenará en el archivo `Bienvenido1.java`.

Error común de programación 2.2

Ocurrirá un error de compilación si el nombre de archivo de una clase `public` *no es exactamente el mismo nombre que el de la clase (tanto por su escritura como por el uso de mayúsculas y minúsculas) seguido de la extensión* `.java`.

Nombres de clases e identificadores

Por convención, todos los nombres de clases comienzan con una letra mayúscula, y la primera letra de cada palabra en el nombre de la clase debe ir en mayúscula (por ejemplo, `EjemploDeNombreDeClase`). El nombre de una clase es un **identificador**, es decir, una serie de caracteres que pueden ser letras, dígitos, guiones bajos (_) y signos de moneda ($), que *no* comiencen con un dígito *ni* tengan espacios. Algunos identificadores válidos son `Bienvenido1`, `$valor`, `_valor`, `m_campoEntrada1` y `boton7`. El nombre `7boton` *no* es un identificador válido, ya que comienza con un dígito, y el nombre `campo entrada` *tampoco* lo es debido a que contiene un espacio. Por lo general, un identificador que no empieza con una letra mayúscula no es el

nombre de una clase. Java es **sensible a mayúsculas y minúsculas**; es decir, las letras mayúsculas y minúsculas son distintas, por lo que `valor` y `Valor` son distintos identificadores (pero ambos son válidos).

Cuerpo de la clase
Una **llave izquierda** (como en la línea 5), `{`, comienza el **cuerpo** de todas las declaraciones de clases. Su correspondiente **llave derecha** (en la línea 11), `}`, debe terminar cada declaración de una clase. Las líneas 6 a 10 tienen sangría.

Buena práctica de programación 2.3
Aplique sangría a todo el cuerpo de la declaración de cada clase, usando un "nivel" de sangría entre la llave izquierda y la llave derecha, las cuales delimitan el cuerpo de la clase. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura. Usamos tres espacios para formar un nivel de sangría; muchos programadores prefieren dos o cuatro espacios. Sea cual sea el formato que utilice, hágalo de manera consistente.

Tip para prevenir errores 2.2
Cuando escriba una llave izquierda de apertura, {, escriba de inmediato la llave derecha de cierre, }; después vuelva a colocar el cursor entre las dos llaves y aplique sangría para empezar a escribir el cuerpo. Esta práctica ayuda a prevenir errores debido a la falta de llaves. Muchos IDE insertan la llave derecha de cierre por usted cuando escribe la llave izquierda de apertura.

Error común de programación 2.3
Es un error de sintaxis no utilizar las llaves por pares.

Buena práctica de programación 2.4
Por lo general los IDE insertan por usted la sangría en el código. También puede usar la tecla Tab *para aplicar sangría al código. Puede configurar cada IDE para especificar el número de espacios insertados cuando oprima la tecla* Tab.

Declaración de un método
La línea 6

```
    // el método main empieza la ejecución de la aplicación en Java
```

es un comentario de fin de línea que indica el propósito de las líneas 7 a 10 del programa. La línea 7

```
    public static void main(String[] args)
```

es el punto de inicio de toda aplicación en Java. Los **paréntesis** después del identificador `main` indican que éste es un bloque de construcción del programa, al cual se le llama **método**. Las declaraciones de clases en Java por lo general contienen uno o más métodos. En una aplicación en Java, sólo uno de esos métodos *debe* llamarse `main` y hay que definirlo como se muestra en la línea 7; de no ser así, la Máquina Virtual de Java (JVM) no ejecutará la aplicación. Los métodos pueden realizar tareas y devolver información una vez que éstas hayan concluido. En la sección 3.2.5 explicaremos el propósito de la palabra clave `static`. La palabra clave `void` indica que este método *no* devolverá ningún tipo de información. Más adelante veremos cómo puede un método devolver información. Por ahora, sólo copie la primera línea de `main` en sus aplicaciones en Java. En la línea 7, las palabras `String[] args` entre paréntesis son una parte requerida de la declaración del método `main`; hablaremos sobre esto en el capítulo 7.

La llave izquierda en la línea 8 comienza el **cuerpo de la declaración del método**. Su correspondiente llave derecha debe terminarlo (línea 10). La línea 9 en el cuerpo del método tiene sangría entre las llaves.

Buena práctica de programación 2.5

Aplique sangría a todo el cuerpo de la declaración de cada método, usando un "nivel" de sangría entre las llaves que delimitan el cuerpo del método. Este formato resalta la estructura del método y ayuda a que su declaración sea más fácil de leer.

Operaciones de salida con `System.out.println`

La línea 9

```
System.out.println("Bienvenido a la programacion en Java!");
```

indica a la computadora que realice una acción; es decir, que imprima los caracteres contenidos entre los signos de comillas dobles (las comillas dobles *no* se muestran en la salida). En conjunto, las comillas dobles y los caracteres entre ellas son una **cadena**, lo que también se conoce como **cadena de caracteres** o **literal de cadena**. El compilador *no* ignora los caracteres de espacio en blanco dentro de las cadenas. Éstas *no* pueden abarcar varias líneas de código.

`System.out` (que usted predefinió) se conoce como el **objeto de salida estándar**. Permite a una aplicación de Java mostrar información en la **ventana de comandos** desde la cual se ejecuta. En versiones recientes de Microsoft Windows, la ventana de comandos es el **Símbolo del sistema**. En UNIX/Linux/Mac OS X, la ventana de comandos se llama **ventana de terminal** o **shell**. Muchos programadores se refieren a la ventana de comandos simplemente como la **línea de comandos**.

El método `System.out.println` muestra (o imprime) una línea de texto en la ventana de comandos. La cadena dentro de los paréntesis en la línea 9 es el **argumento** para el método. Cuando el método `System.out.println` completa su tarea, coloca el cursor de salida (la ubicación donde se mostrará el siguiente carácter) al principio de la siguiente línea de la ventana de comandos. Esto es similar a lo que ocurre cuando un usuario oprime la tecla *Intro*, al escribir en un editor de texto: el cursor aparece al principio de la siguiente línea en el documento.

Toda la línea 9, incluyendo `System.out.println`, el argumento "`Bienvenido a la programacion en Java!`" entre paréntesis y el **punto y coma** (`;`), se conoce como una **instrucción**. Por lo general, un método contiene una o más instrucciones que realizan su tarea. La mayoría de las instrucciones terminan con punto y coma. Cuando se ejecuta la instrucción de la línea 9, muestra el mensaje `Bienvenido a la programacion en Java!` en la ventana de comandos.

Al aprender a programar, algunas veces es conveniente "descomponer" un programa funcional, para que de esta manera pueda familiarizarse con los mensajes de error de sintaxis del compilador. Estos mensajes no siempre indican el problema exacto en el código. Cuando se encuentre con un mensaje de error, éste le dará una idea de qué fue lo que ocasionó el error [intente quitando un punto y coma o una llave en el programa de la figura 2.1, y vuelva a compilarlo para que pueda ver los mensajes de error que se generan debido a esta omisión].

Tip para prevenir errores 2.3

Cuando el compilador reporta un error de sintaxis, éste tal vez no se encuentre en el número de línea indicado por el mensaje de error. Primero verifique la línea en la que se reportó el error; si no encuentra errores en esa línea, verifique varias de las líneas anteriores.

Uso de los comentarios de fin de línea en las llaves derechas para mejorar la legibilidad

Como ayuda para los principiantes en la programación, incluimos un comentario de fin de línea después de una llave derecha de cierre que termina la declaración de un método y después de una llave de cierre que termina la declaración de una clase. Por ejemplo, la línea 10

```
} // fin del método main
```

indica la llave de cierre del método `main`, y la línea 11

```
} // fin de la clase Bienvenido1
```

indica la llave de cierre de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha. Después de este capítulo omitiremos estos comentarios finales.

Compilación y ejecución de su primera aplicación de Java

Ahora estamos listos para compilar y ejecutar nuestro programa. Vamos a suponer que usted utilizará las herramientas de línea de comandos del Kit de Desarrollo de Java y no un IDE. Para ayudarlos a compilar y ejecutar sus programas en un IDE, proporcionamos videos Dive Into® para los IDE populares Eclipse, NetBeans e IntelliJ IDEA. Estos videos se encuentran en el sitio Web del libro:

```
http://www.deitel.com/books/jhtp10
```

Para compilar el programa, abra una ventana de comandos y cambie al directorio en donde está guardado el programa. La mayoría de los sistemas operativos utilizan el comando `cd` para cambiar directorios. Por ejemplo, en Windows el comando

```
cd c:\ejemplos\cap02\fig02_01
```

cambia al directorio `fig02_01`. En UNIX/Linux/Mac OS X, el comando

```
cd ~/ejemplos/cap02/fig02_01
```

cambia al directorio `fig02_01`. Para compilar el programa, escriba

```
javac Bienvenido1.java
```

Si el programa no contiene errores de compilación, este comando crea un nuevo archivo llamado `Bienvenido1.class` (conocido como el **archivo de clase** para `Bienvenido1`), el cual contiene los códigos de bytes de Java, independientes de la plataforma, que representan nuestra aplicación. Cuando utilicemos el comando `java` para ejecutar la aplicación en una plataforma específica, la JVM traducirá estos códigos de bytes en instrucciones que el sistema operativo y el hardware subyacentes puedan comprender.

Error común de programación 2.4

Cuando use `javac`, si recibe un mensaje como "`comando o nombre de archivo incorrecto`," "`javac: comando no encontrado`" o "`'javac' no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable`", entonces su instalación del software de Java no se completó en forma apropiada. Esto indica que la variable de entorno `PATH` del sistema no se estableció de manera apropiada. Consulte las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, después de corregir la variable `PATH`, es probable que necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.

Cada mensaje de error de sintaxis contiene el nombre de archivo y el número de línea en donde ocurrió el error. Por ejemplo, `Bienvenido1.java:6` indica que ocurrió un error en la línea 6 del archivo `Bienvenido1.java`. El resto del mensaje proporciona información acerca del error de sintaxis.

Error común de programación 2.5

El mensaje de error del compilador "`class Bienvenido1 is public, should be declared in a file named Welcome1.java`" indica que el nombre del archivo no coincide con el nombre de la clase `public` en el archivo, o que escribió mal el nombre de la clase al momento de compilarla.

Ejecución de la aplicación `Bienvenido1`

Las siguientes instrucciones asumen que los ejemplos del libro se encuentran en `C:\ejemplos` en Windows o en la carpeta `Documents/ejemplos` de su cuenta de usuario en Linux u OS X. Para ejecutar este programa en una ventana de comandos cambie al directorio que contiene `Bienvenido1.java` (`C:\ejemplos\cap02\fig02_01` en Microsoft Windows o `~/Documents/ejemplos/cap02/fig02_01` en Linux/OS X). A continuación, escriba

```
java Bienvenido1
```

y oprima *Intro*. Este comando inicia la JVM, la cual carga el archivo `Bienvenido1.class`. El comando *omite* la extensión `.class` del nombre de archivo; de lo contrario, la JVM *no* ejecutará el programa. La JVM llama al método `main` de `Bienvenido1`. A continuación, la instrucción de la línea 9 de `main` muestra "`Bienvenido a la programacion en Java!`". La figura 2.2 muestra el programa ejecutándose en una ventana de **Símbolo del sistema** de Microsoft Windows [*nota:* muchos entornos muestran los símbolos del sistema con fondos negros y texto blanco. En nuestro entorno ajustamos esta configuración para que nuestras capturas de pantalla fueran más legibles].

Tip para prevenir errores 2.4

Al tratar de ejecutar un programa en Java, si recibe un mensaje como "`Exception in thread "main" java.lang.NoClassDefFoundError: Bienvenido1`*", quiere decir que su variable de entorno CLASSPATH no está configurada de manera correcta. Consulte las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir un nuevo símbolo del sistema después de configurar la variable CLASSPATH.*

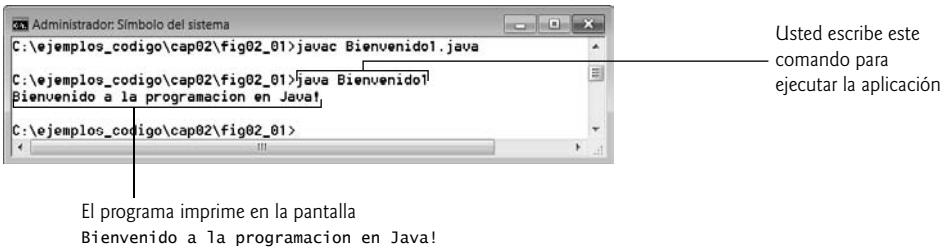

Fig. 2.2 | Ejecución de `Bienvenido1` desde el **Símbolo del sistema**.

2.3 Edición de su primer programa en Java

En esta sección modificaremos el ejemplo de la figura 2.1 para imprimir texto en una línea mediante el uso de varias instrucciones, y para imprimir texto en varias líneas mediante una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

Es posible mostrar la línea de texto `Bienvenido a la programacion en Java!` de varias formas. La clase `Bienvenido2`, que se muestra en la figura 2.3, utiliza dos instrucciones (líneas 9 y 10) para producir el resultado que se muestra en la figura 2.1 [*nota:* de aquí en adelante, resaltaremos las características nuevas y las características clave en cada listado de código, como se muestra en las líneas 9 y 10 de este programa].

Este programa es similar a la figura 2.1, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir una línea de texto con varias instrucciones.
```

```
1  // Fig. 2.3: Bienvenido2.java
2  // Imprimir una línea de texto con varias instrucciones.
3
4  public class Bienvenido2
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
9        System.out.print("Bienvenido a ");
10       System.out.println("la programacion en Java!");
11    } // fin del método main
12 } // fin de la clase Bienvenido2
```

```
Bienvenido a la programacion en Java!
```

Fig. 2.3 | Imprimir una línea de texto con varias instrucciones.

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido2`. Las líneas 9 y 10 del método `main`

```
System.out.print("Bienvenido a ");
System.out.println("la programacion en Java!");
```

muestran una línea de texto. La primera instrucción utiliza el método `print` de `System.out` para mostrar una cadena. Cada instrucción `print` o `println` continúa mostrando caracteres a partir de donde la última instrucción `print` o `println` dejó de mostrarlos. A diferencia de `println`, después de mostrar su argumento, `print` *no* posiciona el cursor de salida al inicio de la siguiente línea en la ventana de comandos; el siguiente carácter que muestra el programa aparecerá *justo después* del último carácter que muestre `print`. Por lo tanto, la línea 10 coloca el primer carácter de su argumento (la letra "l") inmediatamente después del último carácter que muestra la línea 9 (el *carácter de espacio* antes del carácter de comilla doble de cierre de la cadena).

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas mediante el uso de los **caracteres de nueva línea**, los cuales indican a los métodos `print` y `println de System.out` cuándo deben colocar el cursor de salida al inicio de la siguiente línea en la ventana de comandos. Al igual que las líneas en blanco, los espacios y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. El programa de la figura 2.4 muestra cuatro líneas de texto mediante el uso de caracteres de nueva línea para determinar cuándo empezar cada nueva línea. La mayor parte del programa es idéntico a los de las figuras 2.1 y 2.3.

```
1  // Fig. 2.4: Bienvenido3.java
2  // Imprimir varias líneas de texto con una sola instrucción.
3
4  public class Bienvenido3
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main(String[] args)
8     {
```

Fig. 2.4 | Imprimir varias líneas de texto con una sola instrucción (parte 1 de 2).

```
 9       System.out.println("Bienvenido\na\nla programacion\nen Java!");
10    } // fin del método main
11 } // fin de la clase Bienvenido3
```

```
Bienvenido
a
la programacion
en Java!
```

Fig. 2.4 | Imprimir varias líneas de texto con una sola instrucción (parte 2 de 2).

La línea 9

```
System.out.println("Bienvenido\na\nla programacion\nen Java!");
```

muestra cuatro líneas de texto en la ventana de comandos. Por lo general, los caracteres en una cadena se muestran *justo* como aparecen en las comillas dobles. Sin embargo, observe que los dos caracteres \ y n (que se repiten tres veces en la instrucción) *no* aparecen en la pantalla. La **barra diagonal inversa** (\) se conoce como **carácter de escape**, el cual tiene un significado especial para los métodos `print` y `println` de `System.out`. Cuando aparece una barra diagonal inversa en una cadena de caracteres, Java la combina con el siguiente carácter para formar una **secuencia de escape**. La secuencia de escape `\n` representa el carácter de nueva línea. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos.

En la figura 2.5 se listan varias secuencias de escape comunes, con descripciones de cómo afectan la manera de mostrar caracteres en la ventana de comandos. Para obtener una lista completa de secuencias de escape, visite

```
http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10.6.
```

Secuencia de escape	Descripción
`\n`	Nueva línea. Coloca el cursor de la pantalla al inicio de la *siguiente* línea.
`\t`	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
`\r`	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea *actual*; *no* avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro *sobrescribe* los caracteres previamente impresos en esa línea.
`\\`	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
`\"`	Doble comilla. Se usa para imprimir un carácter de doble comilla. Por ejemplo, `System.out.println("\"entre comillas\"");` muestra "`entre comillas`".

Fig. 2.5 | Algunas secuencias de escape comunes.

2.4 Cómo mostrar texto con `printf`

El método `System.out.printf` ("`f`" significa "formato") muestra datos *con formato*. La figura 2.6 usa este método para mostrar en dos líneas las cadenas "`Bienvenido a`" y "`la programacion en Java!`".

```
1  // Fig. 2.6: Bienvenido4.java
2  // Imprimir varias líneas con el método System.out.printf.
3
4  public class Bienvenido4
5  {
6     // el método main empieza la ejecución de la aplicación de Java
7     public static void main(String[] args)
8     {
9        System.out.printf("%s%n%s%n",
10          "Bienvenido a", "la programacion en Java!");
11    } // fin del método main
12 } // fin de la clase Bienvenido4
```

```
Bienvenido a
la programacion en Java!
```

Fig. 2.6 | Imprimir varias líneas de texto con el método `System.out.printf`.

Las líneas 9 y 10

```
System.out.printf("%s%n%s%n",
   "Bienvenido a", "la programacion en Java!");
```

llaman al método `System.out.printf` para mostrar la salida del programa. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos se colocan en una **lista separada por comas**. Al proceso de llamar a un método también se le conoce como **invocar** un método.

Buena práctica de programación 2.6

Coloque un espacio después de cada coma (,) en una lista de argumentos para que sus programas sean más legibles.

Las líneas 9 y 10 representan sólo *una* instrucción. Java permite dividir instrucciones extensas en varias líneas. Aplicamos sangría a la línea 10 para indicar que es la *continuación* de la línea 9.

Error común de programación 2.6

Dividir una instrucción a la mitad de un identificador o de una cadena es un error de sintaxis.

El primer argumento del método `printf` es una **cadena de formato** que puede consistir en **texto fijo** y **especificadores de formato**. El método `printf` imprime el texto fijo de igual forma que `print` o `println`. Cada especificador de formato es un *receptáculo* para un valor y especifica el *tipo de datos* a desplegar. Los especificadores de formato también pueden incluir información de formato opcional.

Los especificadores de formato empiezan con un signo porcentual (%) y van seguidos de un carácter que representa el *tipo de datos*. Por ejemplo, el especificador de formato **`%s`** es un receptáculo para una cadena. La cadena de formato en la línea 9 especifica que `printf` debe desplegar dos cadenas, y que a cada cadena le debe seguir un carácter de nueva línea. En la posición del primer especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. En cada posición posterior del especificador de formato, `printf` sustituye el valor del siguiente argumento. Así, este ejemplo sustituye "`Bienvenido a`" por el primer `%s` y "`la programacion en Java!`" por el segundo `%s`. La salida muestra que se despliegan dos líneas de texto en pantalla.

Cabe mencionar que, en vez de usar la secuencia de escape `\n`, usamos el especificador de formato **`%n`**, el cual es un separador de línea *portable* entre distintos sistemas operativos. No puede usar `%n` en el argu-

mento para `System.out.print` o `System.out.println`; sin embargo, el separador de línea que produce `System.out.println` *después* de mostrar su argumento, *es* portable entre distintos sistemas operativos. El apéndice I en línea presenta más detalles sobre cómo dar formato a la salida con `printf`.

2.5 Otra aplicación: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos **enteros** (números completos, como –22, 7, 0 y 1024) que el usuario introduce mediante el teclado, después calcula la suma de los valores y muestra el resultado. Este programa debe llevar la cuenta de los números que suministra el usuario para los cálculos que el programa realiza posteriormente. Los programas recuerdan números y otros datos en la memoria de la computadora, y acceden a esos datos a través de elementos del programa conocidos como **variables**. El programa de la figura 2.7 demuestra estos conceptos. En la salida de ejemplo, usamos texto en negritas para identificar la entrada del usuario (por ejemplo, **45** y **72**). Como en los programas anteriores, las líneas 1 y 2 indican el número de figura, nombre de archivo y propósito del programa.

```
1  // Fig. 2.7: Suma.java
2  // Programa que recibe dos números y muestra la suma.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class Suma
6  {
7     // el método main empieza la ejecución de la aplicación en Java
8     public static void main(String[] args)
9     {
10       // crea objeto Scanner para obtener la entrada de la ventana de comandos
11       Scanner entrada = new Scanner(System.in);
12
13       int numero1; // primer número a sumar
14       int numero2; // segundo número a sumar
15       int suma; // suma de numero1 y numero2
16
17       System.out.print("Escriba el primer entero: "); // indicador
18       numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20       System.out.print("Escriba el segundo entero: "); // indicador
21       numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23       suma = numero1 + numero2; // suma los números, después almacena el total en suma
24
25       System.out.printf("La suma es %d%n", suma); // muestra la suma
26    } // fin del método main
27 } // fin de la clase Suma
```

```
Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117
```

Fig. 2.7 | Programa que recibe dos números y muestra la suma.

2.5.1 Declaraciones `import`

Una gran fortaleza de Java es su extenso conjunto de clases predefinidas que podemos *reutilizar*, en vez de "reinventar la rueda". Estas clases se agrupan en **paquetes** (*grupos con nombre de clases relacionadas*) y se

conocen en conjunto como la **biblioteca de clases de Java**, o **Interfaz de programación de aplicaciones de Java** (**API de Java**). La línea 3

```
import java.util.Scanner; // el programa usa la clase Scanner
```

es una **declaración `import`** que ayuda al compilador a localizar una clase que se utiliza en este programa. Indica que este ejemplo utiliza la clase `Scanner` predefinida de Java (que veremos en breve) del paquete **`java.util`**. Así, el compilador se asegura de que utilice la clase en forma correcta.

Error común de programación 2.7

Todas las declaraciones `import` deben aparecer antes de la declaración de la primera clase en el archivo. Colocar una declaración `import` dentro del cuerpo de la declaración de una clase, o después de la declaración de la misma, es un error de sintaxis.

Error común de programación 2.8

Si olvida incluir una declaración `import` para una clase que debe importarse, se produce un error de compilación que contiene un mensaje tal como "`cannot find symbol`". Cuando esto ocurra, verifique que haya proporcionado las declaraciones `import` apropiadas y que los nombres en las mismas estén escritos correctamente, incluyendo el uso apropiado de las letras mayúsculas y minúsculas.

Observación de ingeniería de software 2.1

*En cada nueva versión de Java, por lo general las API contienen nuevas herramientas que corrigen errores, mejoran el rendimiento u ofrecen mejores formas de realizar tareas. Las correspondientes versiones anteriores ya no son necesarias, por lo que no deben usarse. Se dice que dichas API están **obsoletas** y podrían retirarse de versiones posteriores de Java.*

A menudo se encontrará con versiones obsoletas de API cuando explore la documentación de las API. El compilador le advertirá cuando compile código que utilice API obsoletas. Si compila su código con `javac` mediante el uso del argumento de línea de comandos `–deprecation`, el compilador le indicará las características obsoletas que está usando. Para cada una, la documentación en línea (`http://docs.oracle.com/javase/7/docs/api/`) indica la característica obsoleta y por lo general contiene vínculos hacia la nueva característica que la sustituye.

2.5.2 Declaración de la clase `Suma`

La línea 5

```
public class Suma
```

empieza la declaración de la clase `Suma`. El nombre de archivo para esta clase `public` debe ser `Suma.java`. Recuerde que el cuerpo de cada declaración de clase empieza con una llave izquierda de apertura (línea 6) y termina con una llave derecha de cierre (línea 27).

La aplicación empieza a ejecutarse con el método `main` (líneas 8 a la 26). La llave izquierda (línea 9) marca el inicio del cuerpo de `main`, y la correspondiente llave derecha (línea 26) marca su final. Al método `main` se le aplica un nivel de sangría en el cuerpo de la clase `Suma`, y al código en el cuerpo de `main` se le aplica otro nivel para mejorar la legibilidad.

2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado

Una **variable** es una ubicación en la memoria de la computadora, en donde se puede guardar un valor para utilizarlo después en un programa. Todas las variables *deben* declararse con un **nombre** y un **tipo** *antes* de poder usarse. El *nombre* de una variable permite al programa acceder al *valor* de la variable en memoria. El

nombre de una variable puede ser cualquier identificador válido; de nuevo, una serie de caracteres compuestos por letras, dígitos, guiones bajos (_) y signos de moneda ($) que *no* comiencen con un dígito y *no* contengan espacios. El *tipo* de una variable especifica el tipo de información que se guarda en esa ubicación de memoria. Al igual que las demás instrucciones, las instrucciones de declaración terminan con punto y coma (;).

La línea 11

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

es una **instrucción de declaración de variable** que especifica el *nombre* (`entrada`) y *tipo* (`Scanner`) de una variable que se utiliza en este programa. Un objeto **`Scanner`** permite a un programa leer datos (por ejemplo: números y cadenas) para usarlos en un programa. Los datos pueden provenir de muchas fuentes, como un archivo en disco o desde el teclado de un usuario. Antes de usar un objeto `Scanner`, hay que crearlo y especificar el *origen* de los datos.

El signo = en la línea 11 indica que es necesario **inicializar** la variable `entrada` tipo `Scanner` (es decir, hay que prepararla para usarla en el programa) en su declaración con el resultado de la expresión a la derecha del signo igual: `new Scanner(System.in)`. Esta expresión usa la palabra clave **`new`** para crear un objeto `Scanner` que lee los datos escritos por el usuario mediante el teclado. El **objeto de entrada estándar**, **`System.in`**, permite a las aplicaciones leer los *bytes* de datos escritos por el usuario. El objeto `Scanner` traduce estos bytes en tipos (como `int`) que se pueden usar en un programa.

2.5.4 Declaración de variables para almacenar enteros

Las instrucciones de declaración de variables en las líneas 13 a la 15

```
int numero1; // primer número a sumar
int numero2; // segundo número a sumar
int suma; // suma de numero1 y numero2
```

declaran que las variables `numero1`, `numero2` y `suma` contienen datos de tipo **`int`**; estas variables pueden contener valores *enteros* (números completos, como `72`, `-1127` y `0`). Estas variables *no* se han inicializado todavía. El rango de valores para un `int` es de -2,147,483,648 a +2,147,483,647 [*nota*: los valores `int` que use en un programa tal vez no contengan comas].

Hay otros tipos de datos como **`float`** y **`double`**, para guardar números reales, y el tipo **`char`**, para guardar datos de caracteres. Los números reales son números que contienen puntos decimales, como `3.4`, `0.0` y `-11.19`. Las variables de tipo `char` representan caracteres individuales, como una letra en mayúscula (Vg. `A`), un dígito (Vg. `7`), un carácter especial (Vg. `*` o `%`) o una secuencia de escape (como el carácter de tabulación, `\t`). Los tipos tales como `int`, `float`, `double` y `char` se conocen como **tipos primitivos**. Los nombres de los tipos primitivos son palabras clave y deben aparecer completamente en minúsculas. El apéndice D sintetiza las características de los ocho tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`).

Es posible declarar varias variables del mismo tipo en una sola declaración, separando con comas los nombres de las variables (es decir, una lista de nombres de variables separados por comas). Por ejemplo, las líneas 13 a la 15 se pueden escribir también así:

```
int numero1, // primer número a sumar
    numero2, // segundo número a sumar
    suma; // suma de numero1 y numero2
```

Buena práctica de programación 2.7

Declare cada variable en su propia declaración. Este formato permite insertar un comentario descriptivo enseguida de cada declaración.

Buena práctica de programación 2.8

Seleccionar nombres de variables significativos ayuda a que un programa se autodocumente (es decir, que sea más fácil entender con sólo leerlo, en lugar de leer la documentación asociada o crear y ver un número excesivo de comentarios).

Buena práctica de programación 2.9

Por convención, los identificadores de nombre de variables empiezan con una letra minúscula, y cada una de las palabras del nombre que van después de la primera, deben empezar con una letra mayúscula. Por ejemplo, el identificador `primerNumero` *tiene una* `N` *mayúscula en su segunda palabra,* `Numero`*. A esta convención de nombres se le conoce como* `CamelCase`*, ya que las mayúsculas sobresalen de forma similar a la joroba de un camello.*

2.5.5 Cómo pedir la entrada al usuario

La línea 17

```
System.out.print("Escriba el primer entero: "); // indicador
```

utiliza `System.out.print` para mostrar el mensaje "`Escriba el primer entero: `". Este mensaje se conoce como **indicador**, ya que indica al usuario que debe realizar una acción específica. En este ejemplo utilizamos el método `print` en vez de `println` para que la entrada del usuario aparezca en la misma línea que la del indicador. En la sección 2.2 vimos que, por lo general, los identificadores que empiezan con letras mayúsculas representan nombres de clases. La clase `System` forma parte del paquete **`java.lang`**. Cabe mencionar que la clase `System` *no* se importa con una declaración `import` al principio del programa.

Observación de ingeniería de software 2.2

El paquete `java.lang` *se importa de manera predeterminada en todos los programas de Java; por ende, las clases en* `java.lang` *son las únicas en la API de Java que no requieren una declaración* `import`*.*

2.5.6 Cómo obtener un valor `int` como entrada del usuario

La línea 18

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

utiliza el método `nextInt` del objeto `entrada` de la clase `Scanner` para obtener un entero del usuario mediante el teclado. En este punto, el programa *espera* a que el usuario escriba el número y oprima *Intro* para enviar el número al programa.

Nuestro programa asume que el usuario escribirá un valor de entero válido. De no ser así, se producirá un error lógico en tiempo de ejecución y el programa terminará. El capítulo 11, Manejo de excepciones: un análisis más detallado, habla sobre cómo hacer sus programas más robustos al permitirles manejar dichos errores. Esto también se conoce como hacer que su programa sea *tolerante a fallas*.

En la línea 18, colocamos el resultado de la llamada al método `nextInt` (un valor `int`) en la variable `numero1` mediante el uso del **operador de asignación**, `=`. La instrucción se lee como "`numero1` *obtiene* el valor de `entrada.nextInt()`". Al operador `=` se le llama **operador binario**, ya que tiene *dos* **operandos**: `numero1` y el resultado de la llamada al método `entrada.nextInt()`. Esta instrucción se llama *instrucción de asignación*, ya que *asigna* un valor a una variable. Todo lo que está a la *derecha* del operador de asignación (`=`) se evalúa siempre *antes* de realizar la asignación.

Buena práctica de programación 2.10

Coloque espacios en ambos lados de un operador binario para mejorar la legibilidad del programa.

2.5.7 Cómo pedir e introducir un segundo `int`

La línea 20

```
System.out.print("Escriba el segundo entero: "); // indicador
```

indica al usuario que escriba el segundo entero. La línea 21

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

lee el segundo entero y lo asigna a la variable `numero2`.

2.5.8 Uso de variables en un cálculo

La línea 23

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación, `=`. La instrucción se lee como "`suma` *obtiene* el valor de `numero1 + numero2`". Cuando el programa encuentra la operación de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un *operador binario*; sus *dos* operandos son las variables `numero1` y `numero2`. Las partes de las instrucciones que contienen cálculos se llaman **expresiones**. De hecho, una expresión es cualquier parte de una instrucción que tiene un *valor* asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión `entrada.nextInt()` es el entero escrito por el usuario.

2.5.9 Cómo mostrar el resultado del cálculo

Una vez realizado el cálculo, la línea 25

```
System.out.printf("La suma es %d%n", suma); // muestra la suma
```

utiliza el método `System.out.printf` para mostrar la `suma`. El especificador de formato `%d` es un receptáculo para un valor `int` (en este caso, el valor de `suma`); la letra `d` se refiere a "entero decimal". El resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `printf` imprime en pantalla "`La suma es` ", seguido del valor de `suma` (en la posición del especificador de formato `%d`) y de una nueva línea.

También es posible realizar cálculos *dentro* de instrucciones `printf`. Podríamos haber combinado las instrucciones de las líneas 23 y 25 en la siguiente instrucción:

```
System.out.printf("La suma es %d%n", (numero1 + numero2));
```

Los paréntesis alrededor de la expresión `numero1 + numero2` son opcionales; se incluyen para enfatizar que el valor de *toda* la expresión se imprime en la posición del especificador de formato `%d`. Se dice que dichos paréntesis son **redundantes**.

2.5.10 Documentación de la API de Java

Para cada nueva clase de la API de Java que utilicemos, hay que indicar el paquete en el que se ubica. Esta información nos ayuda a localizar las descripciones de cada paquete y clase en la documentación de la API de Java. Puede encontrar una versión basada en Web de esta documentación en

```
http://docs.oracle.com/javase/7/docs/api/index.html
```

También puede descargar esta documentación de la sección Additional Resources (Recursos adicionales) en

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

El apéndice F muestra cómo utilizar esta documentación.

2.6 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a ciertas *ubicaciones* en la memoria de la computadora. Toda variable tiene un **nombre**, un **tipo**, un **tamaño** (en bytes) y un **valor**.

En el programa de suma de la figura 2.7, cuando se ejecuta la instrucción (línea 18):

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

el número escrito por el usuario se coloca en una ubicación de memoria que corresponde al nombre `numero1`. Suponga que el usuario escribe `45`. La computadora coloca ese valor entero en la ubicación `numero1` (figura 2.8) y sustituye al valor anterior en esa ubicación (si había uno). El valor anterior se pierde, por lo que se dice que este proceso es *destructivo*.

Fig. 2.8 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

Cuando se ejecuta la instrucción (línea 21)

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

suponga que el usuario escribe `72`. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 2.9.

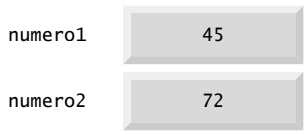

Fig. 2.9 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

Una vez que el programa de la figura 2.7 obtiene valores para `numero1` y `numero2`, los suma y coloca el total en la variable `suma`. La instrucción (línea 23)

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

realiza la suma y después sustituye el valor anterior de `suma`. Una vez que se calcula `suma`, la memoria aparece como se muestra en la figura 2.10. Los valores de `numero1` y `numero2` aparecen exactamente como antes de usarlos en el cálculo de `suma`. Estos valores se utilizaron, pero *no* se destruyeron, cuando la computadora realizó el cálculo. Por ende, cuando se lee un valor de una ubicación de memoria, el proceso es *no destructivo*.

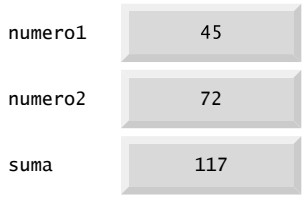

Fig. 2.10 | Ubicaciones de memoria, después de almacenar la suma de `numero1` y `numero2`.

2.7 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 2.11. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El **asterisco** (`*`) indica la multiplicación, y el signo de porcentaje (`%`) es el **operador residuo**, el cual describiremos en breve. Los operadores aritméticos en la figura 2.11 son operadores *binarios*, ya que funcionan con *dos* operandos. Por ejemplo, la expresión `f + 7` contiene el operador binario + junto con los dos operandos `f` y `7`.

Operación en Java	Operador	Expresión algebraica	Expresión en Java
Suma	`+`	$f + 7$	`f + 7`
Resta	`-`	$p - c$	`p - c`
Multiplicación	`*`	bm	`b * m`
División	`/`	x / y o $\frac{x}{y}$ o $x \div y$	`x / y`
Residuo	`%`	$r \bmod s$	`r % s`

Fig. 2.11 | Operadores aritméticos.

La **división de enteros** produce un cociente entero. Por ejemplo, la expresión `7 / 4` da como resultado `1`, y la expresión `17 / 5` da como resultado `3`. Cualquier parte fraccionaria en una división de enteros simplemente se *descarta* (es decir, se *trunca*); no ocurre un *redondeo*. Java proporciona el operador residuo, `%`, el cual produce el residuo después de la división. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce `3`, y `17 % 5` produce `2`. Este operador se utiliza más comúnmente con operandos enteros, pero también puede usarse con otros tipos aritméticos. En los ejercicios de este capítulo y de capítulos posteriores, consideramos muchas aplicaciones interesantes del operador residuo, como la de determinar si un número es múltiplo de otro.

Expresiones aritméticas en formato de línea recta

Las expresiones aritméticas en Java deben escribirse en **formato de línea recta** para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "`a` dividida entre `b`" deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica no es generalmente aceptable para los compiladores:

$$\frac{a}{b}$$

Paréntesis para agrupar subexpresiones
Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar a por la cantidad b + c, escribimos

```
a * (b + c)
```

Si una expresión contiene **paréntesis anidados**, como

```
((a + b) * c)
```

se evalúa *primero* la expresión en el conjunto *más interno* de paréntesis (a + b en este caso).

Reglas de precedencia de operadores
Java aplica los operadores dentro de expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que por lo general son las mismas que las que se utilizan en álgebra:

1. Las operaciones de multiplicación, división y residuo se aplican primero. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
2. Las operaciones de suma y resta se aplican a continuación. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

Estas reglas permiten a Java aplicar los operadores en el *orden* correcto.[1] Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Algunos operadores se asocian de derecha a izquierda. La figura 2.12 sintetiza estas reglas de precedencia de operadores. En el apéndice A se incluye una tabla de precedencias completa.

Operador(es)	Operación(es)	Orden de evaluación (precedencia)
* / %	Multiplicación División Residuo	Se evalúan primero. Si hay varios operadores de este tipo, se evalúan de *izquierda a derecha*.
+ -	Suma Resta	Se evalúan después. Si hay varios operadores de este tipo, se evalúan de *izquierda a derecha*.
=	Asignación	Se evalúa al último.

Fig. 2.12 | Precedencia de los operadores aritméticos.

Ejemplos de expresiones algebraicas y de Java
Ahora consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo lista una expresión algebraica y su equivalente en Java. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

1 Utilizamos ejemplos simples para explicar el *orden de evaluación* de las expresiones. Existen situaciones sutiles que se presentan en las expresiones más complejas que veremos más adelante en el libro. Para obtener más información sobre el orden de evaluación, vea el capítulo 15 de *The Java™ Language Specification* (`http://docs.oracle.com/javase/specs/jls/se7/html/index.html`).

Álgebra: $m = \dfrac{a + b + c + d + e}{5}$

Java:
```
m = (a + b + c + d + e) / 5;
```

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa (`a + b + c + d + e`) va a dividirse entre `5`. Si por error se omiten los paréntesis, obtenemos `a + b + c + d + e / 5`, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

El siguiente es un ejemplo de una ecuación de línea recta:

Álgebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia sobre la suma. La asignación ocurre al último, ya que tiene menor precedencia que la multiplicación o la suma.

El siguiente ejemplo contiene las operaciones residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
    6   1   2   4   3   5
```

Los números dentro de los círculos bajo la instrucción indican el *orden* en el que Java aplica los operadores. Las operaciones `*`, `%` y `/` se evalúan primero, en orden de *izquierda a derecha* (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que `+` y `-`. Las operaciones `+` y `-` se evalúan a continuación. Estas operaciones también se aplican de *izquierda a derecha*. El operador de asignación (`=`) se evalúa al último.

Evaluación de un polinomio de segundo grado

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de una expresión de asignación que incluye un polinomio de segundo grado $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

Las operaciones de multiplicación se evalúan primero en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma (como Java no tiene operador aritmético para los exponentes, x^2 se representa como `x * x`. La sección 5.4 muestra una alternativa para los exponentes). A continuación, se evalúan las operaciones de suma, de *izquierda a derecha*. Suponga que `a`, `b`, `c` y `x` se inicializan (reciben valores) como sigue: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 2.13 muestra el orden en el que se aplican los operadores.

Podemos usar *paréntesis redundantes* (paréntesis innecesarios) para hacer que una expresión sea más clara. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

```
y = (a * x * x) + (b * x) + c;
```

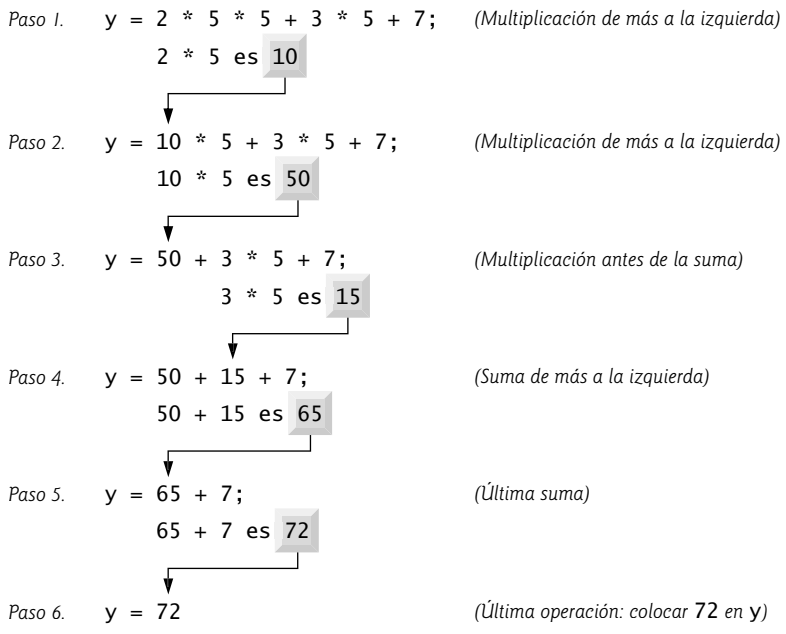

Fig. 2.13 | Orden en el cual se evalúa un polinomio de segundo grado.

2.8 Toma de decisiones: operadores de igualdad y relacionales

Una **condición** es una expresión que puede ser **verdadera** (`true`) o **falsa** (`false`). Esta sección presenta la **instrucción `if` de Java**, la cual permite que un programa tome una **decisión**, con base en el valor de una condición. Por ejemplo, la condición "la calificación es mayor o igual que 60" determina si un estudiante pasó o no una prueba. Si la condición en una instrucción `if` es *verdadera*, se ejecuta el cuerpo de la instrucción `if`. Si la condición es *falsa*, no se ejecuta el cuerpo. Veremos un ejemplo en breve.

Las condiciones en las instrucciones `if` pueden formarse utilizando los **operadores de igualdad** (== y !=) y los **operadores relacionales** (>, <, >= y <=) que se sintetizan en la figura 2.14. Ambos tipos de operadores de igualdad tienen el mismo nivel de precedencia, que es *menor* que la precedencia de los operadores relacionales. Los operadores de igualdad se asocian de *izquierda* a *derecha*. Todos los operadores relacionales tienen el mismo nivel de precedencia y también se asocian de *izquierda* a *derecha*.

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores de igualdad			
=	==	`x == y`	x es igual que y
≠	!=	`x != y`	x no es igual que y

Fig. 2.14 | Operadores de igualdad y relacionales (parte 1 de 2).

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Fig. 2.14 | Operadores de igualdad y relacionales (parte 2 de 2).

En la figura 2.15 se utilizan seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es *verdadera*, se ejecuta la instrucción asociada con esa instrucción `if`; en caso contrario, se omite la instrucción. Utilizamos un objeto `Scanner` para recibir los dos enteros del usuario y almacenarlos en las variables `numero1` y `numero2`. Después, el programa *compara* los números y muestra los resultados de las comparaciones que son verdaderas.

```
1  // Fig. 2.15: Comparacion.java
2  // Compara enteros utilizando instrucciones if, operadores relacionales
3  // y de igualdad.
4  import java.util.Scanner; // el programa utiliza la clase Scanner
5
6  public class Comparacion
7  {
8     // el método main empieza la ejecución de la aplicación en Java
9     public static void main(String[] args)
10    {
11       // crea objeto Scanner para obtener la entrada de la ventana de comandos
12       Scanner entrada = new Scanner(System.in);
13
14       int numero1; // primer número a comparar
15       int numero2; // segundo número a comparar
16
17       System.out.print("Escriba el primer entero: "); // indicador
18       numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20       System.out.print("Escriba el segundo entero: "); // indicador
21       numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23       if (numero1 == numero2)
24          System.out.printf("%d == %d%n", numero1, numero2);
25
26       if (numero1 != numero2)
27          System.out.printf("%d != %d%n", numero1, numero2);
28
29       if (numero1 < numero2)
30          System.out.printf("%d < %d%n", numero1, numero2);
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 1 de 2).

```
31
32       if (numero1 > numero2)
33          System.out.printf("%d > %d%n", numero1, numero2);
34
35       if (numero1 <= numero2)
36          System.out.printf("%d <= %d%n", numero1, numero2);
37
38       if (numero1 >= numero2)
39          System.out.printf("%d >= %d%n", numero1, numero2);
40    } // fin del método main
41 } // fin de la clase Comparacion
```

```
Escriba el primer entero: 777
Escriba el segundo entero: 777
777 == 777
777 <= 777
777 >= 777
```

```
Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 2 de 2).

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `main` de la clase (líneas 9 a 40) empieza la ejecución del programa. La línea 12

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

declara la variable entrada de la clase `Scanner` y le asigna un objeto `Scanner` que recibe datos de la entrada estándar (es decir, del teclado).

Las líneas 14 y 15

```
int numero1; // primer número a comparar
int numero2; // segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 17-18

```
System.out.print("Escriba el primer entero: "); // indicador
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

piden al usuario que introduzca el primer entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero1`.

Las líneas 20-21

```
System.out.print("Escriba el segundo entero: "); // indicador
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

piden al usuario que introduzca el segundo entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero2`.

Las líneas 23-24

```
if (numero1 == numero2)
   System.out.printf("%d == %d%n", numero1, numero2);
```

compara los valores de las variables `numero1` y `numero2`, para determinar si son iguales o no. Una instrucción `if` siempre empieza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo, pero puede contener varias instrucciones si se encierran entre un conjunto de llaves (`{}`). La sangría de la instrucción del cuerpo que se muestra aquí no es obligatoria, pero mejora la legibilidad del programa al enfatizar que la instrucción en la línea 24 *forma parte de* la instrucción `if` que empieza en la línea 23. La línea 24 sólo se ejecuta si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, si la condición es *verdadera*). Las instrucciones `if` en las líneas 26-27, 29-30, 32-33, 35-36 y 38-39 comparan a `numero1` y `numero2` usando los operadores `!=`, `<`, `>`, `<=` y `>=`, respectivamente. Si la condición en una o más de las instrucciones `if` es verdadera, se ejecuta la instrucción del cuerpo correspondiente.

Error común de programación 2.9

Confundir el operador de igualdad (==) con el de asignación (=) puede producir un error lógico o de compilación. El operador de igualdad debe leerse como "es igual a", y el de asignación como "obtiene" u "obtiene el valor de". Para evitar confusión, algunas personas leen el operador de igualdad como "doble igual" o "igual igual".

Buena práctica de programación 2.11

Al colocar sólo una instrucción por línea en un programa, mejora su legibilidad.

No hay punto y coma (`;`) al final de la primera línea de cada instrucción `if`. Dicho punto y coma produciría un error lógico en tiempo de ejecución. Por ejemplo,

```
if (numero1 == numero2); // error lógico
   System.out.printf("%d == %d%n", numero1, numero2);
```

sería interpretada por Java de la siguiente manera:

```
if (numero1 == numero2)
   ; // instrucción vacía
System.out.printf("%d == %d%n", numero1, numero2);
```

donde el punto y coma que aparece por sí solo en una línea (que se conoce como **instrucción vacía**) es la instrucción que se va a ejecutar si la condición en la instrucción `if` es *verdadera*. Al ejecutarse la instrucción vacía, no se lleva a cabo ninguna tarea. Después el programa continúa con la instrucción de salida, que *siempre* se ejecutaría, *sin importar* que la condición sea verdadera o falsa, ya que la instrucción de salida no forma parte de la instrucción `if`.

Espacio en blanco

Observe el uso del espacio en blanco en la figura 2.15. Recuerde que el compilador casi siempre ignora los caracteres de espacio en blanco. Por lo tanto, las instrucciones pueden dividirse en varias líneas y pueden espaciarse de acuerdo con las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Idealmente las instrucciones deben mantenerse lo más reducidas que sea posible, pero no siempre se puede hacer esto.

Tip para prevenir errores 2.5

Una instrucción larga puede repartirse en varias líneas. Si una sola instrucción debe dividirse en varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si una instrucción se divide en dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

Operadores descritos hasta ahora

La figura 2.16 muestra los operadores que hemos visto hasta ahora, en orden decreciente de precedencia. Todos, con la excepción del operador de asignación, `=`, se asocian de *izquierda* a *derecha*. El operador de asignación, `=`, se asocia de *derecha a izquierda*. El valor de una expresión de asignación es el que se haya asignado a la variable del lado izquierdo del operador `=` (por ejemplo, el valor de la expresión `x = 7` es `7`). Entonces, una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = (y = 0)`, en donde primero se asigna el valor `0` a la variable `y`, y después se asigna el resultado de esa asignación, `0`, a `x`.

Operadores	Asociatividad	Tipo
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	suma
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`=`	derecha a izquierda	asignación

Fig. 2.16 | Precedencia y asociatividad de los operadores descritos hasta ahora.

Buena práctica de programación 2.12

Cuando escriba expresiones que contengan muchos operadores, consulte la tabla de precedencia de operadores (apéndice A). Confirme que las operaciones en la expresión se realicen en el orden que usted espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzarlo, en la misma forma que lo haría con las expresiones algebraicas.

2.9 Conclusión

En este capítulo aprendió sobre muchas características importantes de Java, como mostrar datos en la pantalla en un `Símbolo del sistema`, introducir datos mediante el teclado, realizar cálculos y tomar decisiones. Mediante las aplicaciones que vimos en este capítulo, le presentamos muchos conceptos básicos de programación. Como veremos en el capítulo 3, por lo general las aplicaciones de Java contienen sólo unas cuantas líneas de código en el método `main`, ya que casi siempre estas instrucciones crean los objetos que realizan el trabajo de la aplicación. En el capítulo 3 aprenderá a implementar sus propias clases y a usar objetos de esas clases en las aplicaciones.