Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

14

Cadenas, caracteres y expresiones regulares

El principal defecto del rey Enrique era romper con los dientes, pequeños trozos de hilo.

—Hilaire Belloc

La escritura vigorosa es concisa. Una oración no debe contener palabras innecesarias; un párrafo no debe contener oraciones innecesarias.

—William Strunk, Jr.

He extendido esta carta más de lo usual, ya que carezco de tiempo para hacerla breve.

—Blaise Pascal

Objetivos

En este capítulo aprenderá:

- A crear y manipular objetos cadenas de caracteres inmutables de la clase `String`.
- A crear y manipular objetos cadenas de caracteres mutables de la clase `StringBuilder`.
- A crear y manipular objetos de la clase `Character`.
- A descomponer un objeto `String` en tokens usando el método `split` de `String`.
- A usar expresiones regulares para validar los datos `String` que se introducen en una aplicación.

Plan general

14.1 Introducción

En este capítulo presentamos las herramientas para el procesamiento de cadenas y caracteres en Java. Las técnicas que se describen aquí son apropiadas para validar la entrada en los programas, para mostrar información a los usuarios y para otras manipulaciones basadas en texto. Estas técnicas también son apropiadas para desarrollar editores de texto, procesadores de palabras, software para el diseño de páginas, sistemas de composición computarizados y otros tipos de software para el procesamiento de texto. Ya hemos presentado varias herramientas para el procesamiento de texto en capítulos anteriores. En este capítulo describiremos con detalle las herramientas de las clases `String`, `StringBuilder` y `Character` del paquete `java.lang`. Estas clases proporcionan la base para la manipulación de cadenas y caracteres en Java.

En este capítulo también hablaremos sobre las expresiones regulares que proporcionan a las aplicaciones la capacidad de validar los datos de entrada. Esta funcionalidad se encuentra en la clase `String`, junto con las clases `Matcher` y `Pattern` ubicadas en el paquete `java.util.regex`.

14.2 Fundamentos de los caracteres y las cadenas

Los caracteres son los bloques de construcción básicos de los programas fuente de Java. Todo programa está compuesto de una secuencia de caracteres que cuando se agrupan en forma significativa son interpretados por la computadora como una serie de instrucciones utilizadas para realizar una tarea. Un programa puede contener **literales de carácter**. Una literal de carácter es un valor entero representado como un carácter entre comillas simples. Por ejemplo, '`z`' representa el valor entero de `z`, y '`\n`' representa el valor entero de una nueva línea. El valor de una literal de carácter es el valor entero del carácter en el **conjunto de caracteres Unicode**. En el apéndice B se presentan los equivalentes enteros de los caracteres en el conjunto de caracteres ASCII, el cual es un subconjunto de Unicode (que veremos en el apéndice H).

Recuerde que en la sección 2.2 vimos que una cadena es una secuencia de caracteres que se trata como una sola unidad. Una cadena puede incluir letras, dígitos y varios **caracteres especiales**, tales como `+`, `-`, `*`, `/` y `$`. Una cadena es un objeto de la clase `String`. Las **literales de cadena** (que se almacenan en memoria como objetos `String`) se escriben como una secuencia de caracteres entre comillas dobles, como en:

```
"Juan E. Pérez"              (un nombre)
"Calle Principal 9999"       (una dirección)
"Monterrey, Nuevo León"      (una ciudad y un estado)
"(201) 555-1212"             (un número telefónico)
```

Una cadena puede asignarse a una referencia `String`. La declaración

```
String color = "azul";
```

inicializa la variable `String` de nombre `color` para que haga referencia a un objeto `String` que contiene la cadena "`azul`".

Tip de rendimiento 14.1

Para conservar memoria, Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String` que tiene muchas referencias.

14.3 La clase `String`

La clase `String` se utiliza para representar cadenas en Java. En las siguientes subsecciones cubriremos muchas de las herramientas de la clase `String`.

14.3.1 Constructores de `String`

La clase `String` proporciona constructores para inicializar objetos `String` de varias formas. Cuatro de los constructores se muestran en el método `main` de la figura 14.1.

```
1  // Fig. 14.1: ConstructoresString.java
2  // Constructores de la clase String.
3
4  public class ConstructoresString
5  {
6     public static void main(String[] args)
7     {
8        char[] arregloChar = {'c', 'u', 'm', 'p', 'l', 'e' ' ', 'a', 'n', 'i',
         'o', 's'};
9        String s = new String("hola");
10
11       // usa los constructores de String
12       String s1 = new String();
13       String s2 = new String(s);
14       String s3 = new String(arregloChar);
15       String s4 = new String(arregloChar, 6, 3);
16
17       System.out.printf(
18          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
19    }
20 } // fin de la clase ConstructoresString
```

```
s1 =
s2 = hola
s3 = cumple anios
s4 = día
```

Fig. 14.1 | Constructores de la clase `String`.

En la línea 12 se crea una instancia de un nuevo objeto `String`, utilizando el constructor sin argumentos de la clase `String`, y se le asigna su referencia a `s1`. El nuevo objeto `String` no contiene caracteres (es una **cadena vacía**, que se puede representar también como "") y tiene una longitud de 0. En la línea 13 se crea una instancia de un nuevo objeto `String` utilizando el constructor de la clase `String` que toma un objeto `String` como argumento y se le asigna su referencia a `s2`. El nuevo objeto `String` contiene la misma secuencia de caracteres que el objeto `String` de nombre `s`, el cual se pasa como argumento para el constructor.

Tip de rendimiento 14.2

No es necesario copiar un objeto `String` existente. Los objetos `String` son inmutables ya que la clase `String` no proporciona métodos que permitan modificar el contenido de un objeto `String` después de crearlo.

En la línea 14 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s3`, utilizando el constructor de la clase `String` que toma un arreglo de caracteres como argumento. El nuevo objeto `String` contiene una copia de los caracteres en el arreglo.

En la línea 15 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s4`, utilizando el constructor de la clase `String` que toma un arreglo `char` y dos enteros como argumentos. El segundo argumento especifica la posición inicial (el *desplazamiento*) a partir del cual se accede a los caracteres en el arreglo. Recuerde que el primer carácter se encuentra en la posición `0`. El tercer argumento especifica el número de caracteres (la cuenta) que se van a utilizar del arreglo. El nuevo objeto `String` se forma a partir de los caracteres utilizados. Si el desplazamiento o la cuenta especificados como argumentos ocasionan que se acceda a un elemento fuera de los límites del arreglo de caracteres, se lanza una excepción `StringIndexOutOfBoundsException`.

14.3.2 Métodos `length`, `charAt` y `getChars` de `String`

Los métodos **`length`**, **`charAt`**, y **`getChars`** de `String` devuelven la longitud de un objeto `String`, obtienen el carácter que se encuentra en una ubicación específica de un objeto `String` y recuperan un conjunto de caracteres en un objeto `String` como un arreglo `char`, respectivamente. La figura 14.2 demuestra cada uno de estos métodos.

```
1  // Fig. 14.2: VariosString.java
2  // Esta aplicación muestra los métodos length, charAt y getChars
3  // de la clase String.
4
5  public class VariosString
6  {
7     public static void main(String[] args)
8     {
9        String s1 = "hola a todos";
10       char[] arregloChar = new char[5];
11
12       System.out.printf("s1: %s", s1);
13
14       // prueba el método length
15       System.out.printf("%nLongitud de s1: %d", s1.length());
16
17       // itera a través de los caracteres en s1 con charAt y muestra la cadena
            invertida
18       System.out.printf("%nLa cadena invertida es: ");
```

Fig. 14.2 | Los métodos `length`, `charAt` y `getChars` de `String` (parte 1 de 2).

```
19
20          for (int cuenta = s1.length() - 1; cuenta >= 0; cuenta--)
21             System.out.printf("%c ", s1.charAt(cuenta));
22
23          // copia los caracteres de la cadena a arregloChar
24          s1.getChars(0, 5, arregloChar, 0);
25          System.out.printf("%nEl arreglo de caracteres es: ");
26
27          for (char caracter : charArray)
28             System.out.print(caracter);
29
30          System.out.println();
31       }
32    } // fin de la clase VariosString
```

```
s1: hola a todos
Longitud de s1: 12
La cadena invertida es: s o d o t   a   a l o h
El arreglo de caracteres es: hola
```

Fig. 14.2 | Los métodos `length`, `charAt` y `getChars` de `String` (parte 2 de 2).

En la línea 15 se utiliza el método `length` de `String` para determinar el número de caracteres en la cadena `s1`. Al igual que los arreglos, las cadenas conocen su propia longitud. Sin embargo, a diferencia de los arreglos, para acceder a la longitud de un objeto `String` se usa el método `length` de la clase `String`.

En las líneas 20 y 21 se imprimen los caracteres de la cadena `s1` en orden inverso (y separados por espacios). El método `charAt` de `String` (línea 21) devuelve el carácter ubicado en una posición específica en el objeto `String`. El método `charAt` recibe un argumento entero que se utiliza como el índice, y devuelve el carácter en esa posición. Al igual que los arreglos, se considera que el primer elemento de un objeto `String` está en la posición 0.

En la línea 24 se utiliza el método `getChars` de `String` para copiar los caracteres de un objeto `String` en un arreglo de caracteres. El primer argumento es el índice inicial en la cadena, a partir del cual se van a copiar los caracteres. El segundo argumento es el índice que está una posición más adelante del último carácter que se va a copiar del objeto `String`. El tercer argumento es el arreglo de caracteres en el que se van a copiar los caracteres. El último argumento es el índice inicial en donde se van a colocar los caracteres copiados en el arreglo de caracteres de destino. A continuación, en las líneas 27 y 28 se imprime el contenido del arreglo `char`, un carácter a la vez.

14.3.3 Comparación entre cadenas

En el capítulo 19 hablaremos sobre el ordenamiento y la búsqueda en los arreglos. A menudo la información que se va a ordenar o buscar consiste en cadenas que deben compararse para determinar el orden o para determinar si una cadena aparece en un arreglo (u otra colección). La clase `String` proporciona varios métodos para *comparar* cadenas, como se demuestra en los siguientes dos ejemplos.

Para comprender lo que significa que una cadena sea mayor o menor que otra, considere el proceso de ordenar alfabéticamente una serie de apellidos. Sin duda usted colocaría a "Jones" antes que "Smith", ya que en el alfabeto la primera letra de "Jones" viene antes que la primera letra de "Smith". Pero el alfabeto es algo más que una lista de 26 letras; es una lista *ordenada* de caracteres. Cada letra ocupa una posición específica dentro de la lista. Z es más que una letra del alfabeto; es en específico la letra número veintiséis del alfabeto.

¿Cómo sabe la computadora que una letra "va antes" que otra? Todos los caracteres se representan en la computadora como códigos numéricos (vea el apéndice B). Cuando la computadora compara objetos `String`, en realidad compara los códigos numéricos de los caracteres en estos objetos.

En la figura 14.3 se muestran los métodos `equals`, `equalsIgnoreCase`, `compareTo` y **`regionMatches`** de `String`, y se muestra el uso del operador de igualdad == para comparar objetos `String`.

```
1  // Fig. 14.3: CompararCadenas.java
2  // Los métodos equals, equalsIgnoreCase, compareTo y regionMatches de String.
3
4  public class CompararCadenas
5  {
6     public static void main(String[] args)
7     {
8        String s1 = new String("hola"); // s1 es una copia de "hola"
9        String s2 = "adios";
10       String s3 = "Feliz cumpleanios";
11       String s4 = "feliz cumpleanios";
12
13       System.out.printf(
14          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);
15
16       // prueba la igualdad
17       if (s1.equals("hola"))  // true
18          System.out.println("s1 es igual a \"hola\"");
19       else
20          System.out.println("s1 no es igual a \"hola\"");
21
22       // prueba la igualdad con ==
23       if (s1 == "hola")  // false; no son el mismo objeto
24          System.out.println("s1 es el mismo objeto que \"hola\"");
25       else
26          System.out.println("s1 no es el mismo objeto que \"hola\"");
27
28       // prueba la igualdad (ignora el uso de mayúsculas/minúsculas)
29       if (s3.equalsIgnoreCase(s4))  // true
30          System.out.printf("%s es igual a %s si se ignora el uso de mayusculas/
            minusculas%n", s3, s4);
31       else
32          System.out.println("s3 no es igual a s4");
33
34       // prueba con compareTo
35       System.out.printf(
36          "%ns1.compareTo(s2) es %d", s1.compareTo(s2));
37       System.out.printf(
38          "%ns2.compareTo(s1) es %d", s2.compareTo(s1));
39       System.out.printf(
40          "%ns1.compareTo(s1) es %d", s1.compareTo(s1));
41       System.out.printf(
42          "%ns3.compareTo(s4) es %d", s3.compareTo(s4));
43       System.out.printf(
44          "%ns4.compareTo(s3) es %d%n%n", s4.compareTo(s3));
```

Fig. 14.3 | Los métodos `equals`, `equalsIgnoreCase`, `compareTo` y `regionMatches` de `String` (parte 1 de 2).

```
45
46          // prueba con regionMatches (sensible a mayúsculas/minúsculas)
47          if (s3.regionMatches(0, s4, 0, 5) )
48             System.out.println("Los primeros 5 caracteres de s3 y s4 coinciden");
49          else
50             System.out.println(
51                "Los primeros 5 caracteres de s3 y s4 no coinciden");
52
53          // prueba con regionMatches (ignora el uso de mayúsculas/minúsculas)
54          if (s3.regionMatches(true, 0, s4, 0, 5) )
55             System.out.println(
56                "Los primeros 5 caracteres de s3 y s4 coinciden ignorando uso de
                  mayusculas/minusculas");
57          else
58             System.out.println(
59                "Los primeros 5 caracteres de s3 y s4 no coinciden ignorando uso de
                  mayusculas/minusculas");
60       }
61    } // fin de la clase CompararCadenas
```

```
s1 = hola
s2 = adios
s3 = Feliz cumpleanios
s4 = feliz cumpleanios

s1 es igual a "hola"
s1 no es el mismo objeto que "hola"
Feliz cumpleanios es igual a feliz cumpleanios si se ignora el uso de mayusculas/
minusculas

s1.compareTo(s2) es 1
s2.compareTo(s1) es -1
s1.compareTo(s1) es 0
s3.compareTo(s4) es -32
s4.compareTo(s3) es 32

Los primeros 5 caracteres de s3 y s4 no coinciden
Los primeros 5 caracteres de s3 y s4 coinciden ignorando uso de mayusculas/minusculas
```

Fig. 14.3 | Los métodos `equals`, `equalsIgnoreCase`, `compareTo` y `regionMatches` de `String` (parte 2 de 2).

Método `equals` de `String`

La condición en la línea 17 utiliza el método `equals` para comparar la igualdad entre el objeto `String s1` y la literal `String "hola"`. El método `equals` (un método de la clase `Object`, sobrescrito en `String`) prueba la igualdad entre dos objetos; es decir, que las cadenas contenidas en los dos objetos sean *idénticas*. El método devuelve `true` si el contenido de los objetos es igual y `false` en caso contrario. La condición anterior es `true`, ya que el objeto `String s1` se inicializó con la literal de cadena "`hola`". El método `equals` utiliza una **comparación lexicográfica**; se comparan los valores enteros Unicode (vea el apéndice H para obtener más información) que representan a cada uno de los caracteres en cada objeto `String`. Por lo tanto, si la cadena "`hola`" se compara con la cadena "`HOLA`", el resultado es `false` ya que la representación entera de una letra minúscula es *distinta* de la letra mayúscula correspondiente.

Comparación de objetos String con el operador ==

La condición en la línea 23 utiliza el operador de igualdad == para comparar la igualdad entre el objeto `String s1` y la literal `String` "`hola`". Cuando se comparan valores de tipos primitivos con `==`, el resultado es `true` si *ambos valores son idénticos*. Cuando se comparan referencias con ==, el resultado es `true` si *ambas referencias se refieren al mismo objeto en memoria*. Para comparar la igualdad del contenido en sí (o la información sobre el estado) de los objetos, hay que invocar un método. En el caso de los objetos `String` ese método es `equals`. La condición anterior se evalúa como `false` en la línea 23, ya que la referencia `s1` se inicializó con la instrucción

```
s1 = new String("hola");
```

con lo cual se crea un nuevo objeto `String` con una copia de la literal de cadena "`hola`", y se asigna el nuevo objeto a la variable `s1`. Si `s1` se hubiera inicializado con la instrucción

```
s1 = "hola";
```

que asigna directamente la literal de cadena "`hola`" a la variable `s1`, la condición habría sido `true`. Recuerde que Java trata a todos los objetos de literal de cadena con el mismo contenido como un objeto `String`, al cual puede haber muchas referencias. Por lo tanto, en las líneas 8, 17 y 23 se hace referencia al mismo objeto `String` "`hola`" en memoria.

Error común de programación 14.1

Al comparar referencias con == se pueden producir errores lógicos, ya que == compara las referencias para determinar si se refieren al mismo objeto, *no si dos objetos tienen el* mismo contenido. *Si se comparan dos objetos separados (que contienen los mismos valores) con ==, el resultado será* `false`. *Si va a comparar objetos para determinar si tienen el mismo contenido, utilice el método* `equals`.

Método equalsIgnoreCase de String

Si va a ordenar objetos `String`, puede comparar si son iguales con el método `equalsIgnoreCase`, el cual ignora si las letras en cada objeto `String` son mayúsculas o minúsculas al realizar la comparación. Por lo tanto, la cadena "`hola`" y la cadena "`HOLA`" se consideran iguales. En la línea 29 se utiliza el método `equalsIgnoreCase` de `String` para comparar si el objeto `String s3` (`Feliz Cumpleanios`) es igual a la cadena `s4` (`feliz cumpleanios`). El resultado de esta comparación es `true`, ya que la comparación ignora el uso de mayúsculas y minúsculas.

Método compareTo de String

En las líneas 35 a 44 se utiliza el método `compareTo` de `String` para comparar objetos `String`. El método `compareTo` se declara en la interfaz `Comparable` y se implementa en la clase `String`. En la línea 36 se compara el objeto `String s1` con el objeto `String s2`. El método `compareTo` devuelve `0` si los objetos `String` son iguales, devuelve un número negativo si el objeto `String` que invoca a `compareTo` es menor que el objeto `String` que se pasa como argumento, o devuelve un número positivo si el objeto `String` que invoca a `compareTo` es mayor que la cadena que se pasa como argumento. El método `compareTo` utiliza una comparación *lexicográfica*: compara los valores numéricos de los caracteres correspondientes en cada objeto `String`.

Método regionMatches de String

La condición en la línea 47 utiliza el método `regionMatches` de `String` para comparar si ciertas porciones de dos objetos `String` son iguales. El primer argumento es el índice inicial en el objeto `String` que invoca al método. El segundo argumento es un objeto `String` de comparación. El tercer argumento es

El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#,Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas…

VISITA

www.detodoprogramacion.org

www.detodopython.com

Detodo Python.com

el índice inicial en el objeto `String` de comparación. El último argumento es el número de caracteres a comparar entre los dos objetos `String`. El método devuelve `true` solamente si el número especificado de caracteres son lexicográficamente iguales.

Por último, la condición en la línea 54 utiliza una versión con cinco argumentos del método `regionMatches` de `String` para comparar si ciertas porciones de dos objetos `String` son iguales. Cuando el primer argumento es `true`, el método ignora el uso de mayúsculas y minúsculas en los caracteres que se van a comparar. Los argumentos restantes son idénticos a los que se describieron para el método `regionMatches` con cuatro argumentos.

Métodos `startsWith` y `endsWith` de `String`

En el siguiente ejemplo (figura 14.4) vamos a mostrar los métodos **`startsWith`** y **`endsWith`** de la clase `String`. El método `main` crea el arreglo `cadenas` que contiene las cadenas "`empezo`", "`empezando`", "`termino`" y "`terminando`". El resto del método `main` consiste en tres instrucciones `for` que prueban los elementos del arreglo para determinar si empiezan o terminan con un conjunto específico de caracteres.

```
1  // Fig. 14.4: CadenaInicioFin.java
2  // Los métodos startsWith y endsWith de String.
3
4  public class CadenaInicioFin
5  {
6     public static void main(String[] args)
7     {
8        String[] cadenas = {"empezo", "empezando", "termino", "terminando"};
9
10       // prueba el método startsWith
11       for (String cadena : cadenas)
12       {
13          if (cadena.startsWith("em"))
14             System.out.printf("\"%s\" empieza con \"em\"%n", cadena);
15       }
16
17       System.out.println();
18
19       // prueba el método startsWith empezando desde la posición 2 de la cadena
20       for (String cadena : cadenas)
21       {
22          if (cadena.startsWith("pez", 2))
23             System.out.printf(
24                "\"%s\" empieza con \"pez\" en la posicion 2%n", cadena);
25       }
26
27       System.out.println();
28
29       // prueba el método endsWith
30       for (String cadena : cadenas)
31       {
32          if (cadena.endsWith("do"))
33             System.out.printf("\"%s\" termina con \"do\"%n", cadena);
34       }
35    }
36 } // fin de la clase CadenaInicioFin
```

Fig. 14.4 | Métodos `startsWith` y `endsWith` de la clase `String` (parte 1 de 2).

```
"empezo" empieza con "em"
"empezando" empieza con "em"

"empezo" empieza con "pez" en la posicion 2
"empezando" empieza con "pez" en la posicion 2

"empezando" termina con "do"
"terminando" termina con "do"
```

Fig. 14.4 | Métodos `startsWith` y `endsWith` de la clase `String` (parte 2 de 2).

En las líneas 11 a 15 se utiliza la versión del método `startsWith` que recibe un argumento `String`. La condición en la instrucción `if` (línea 13) determina si cada objeto `String` en el arreglo empieza con los caracteres "`em`". De ser así, el método devuelve `true` y la aplicación imprime ese objeto `String`. En caso contrario, el método devuelve `false` y no ocurre nada.

En las líneas 20 a 25 se utiliza el método `startsWith` que recibe un objeto `String` y un entero como argumentos. El argumento entero especifica el índice en el que debe empezar la comparación en el objeto `String`. La condición en la instrucción `if` (línea 22) determina si cada objeto `String` en el arreglo tiene los caracteres "`pez`", empezando con el tercer carácter en cada objeto `String`. De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

La tercera instrucción `for` (líneas 30 a 34) utiliza el método `endsWith` que recibe un argumento `String`. La condición en la línea 32 determina si cada objeto `String` en el arreglo termina con los caracteres "`do`". De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

14.3.4 Localización de caracteres y subcadenas en las cadenas

A menudo es útil buscar un carácter o conjunto de caracteres en una cadena. Por ejemplo, si usted va a crear su propio procesador de palabras, tal vez quiera proporcionar una herramienta para buscar a través de los documentos. En la figura 14.5 se muestran las diversas versiones de los métodos **`indexOf`** y **`lastIndexOf`** de `String`, que buscan un carácter o subcadena especificados en un objeto `String`.

```
1  // Fig. 14.5: MetodosIndexString.java
2  // Métodos indexOf y lastIndexOf para buscar en cadenas.
3
4  public class MetodosIndexString
5  {
6     public static void main(String[] args)
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba indexOf para localizar un carácter en una cadena
11       System.out.printf(
12          "'c' se encuentra en el indice %d%n", letras.indexOf('c'));
13       System.out.printf(
14          "'a' se encuentra en el indice %d%n", letras.indexOf('a', 1));
15       System.out.printf(
16          "'$' se encuentra en el indice %d%n%n", letras.indexOf('$'));
17
```

Fig. 14.5 | Métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String` (parte 1 de 2).

```
18         // prueba lastIndexOf para buscar un carácter en una cadena
19         System.out.printf("La ultima 'c' se encuentra en el indice %d%n",
20            letras.lastIndexOf('c'));
21         System.out.printf("La ultima 'a' se encuentra en el indice %d%n",
22            letras.lastIndexOf('a', 25));
23         System.out.printf("La ultima '$' se encuentra en el indice %d%n%n",
24            letras.lastIndexOf('$'));
25
26         // prueba indexOf para localizar una subcadena en una cadena
27         System.out.printf("\"def\" se encuentra en el indice %d%n",
28            letras.indexOf("def"));
29         System.out.printf("\"def\" se encuentra en el indice %d%n",
30            letras.indexOf("def", 7));
31         System.out.printf("\"hola\" se encuentra en el indice %d%n%n",
32            letras.indexOf("hola"));
33
34         // prueba lastIndexOf para buscar una subcadena en una cadena
35         System.out.printf("La ultima ocurrencia de \"def\" se encuentra en el
              indice %d%n",
36            letras.lastIndexOf("def"));
37         System.out.printf("La ultima ocurrencia de \"def\" se encuentra en el
              indice %d%n",
38            letras.lastIndexOf("def", 25));
39         System.out.printf("La ultima ocurrencia de \"hola\" se encuentra en el
              indice %d%n",
40            letras.lastIndexOf("hola"));
41      }
42   } // fin de la clase MetodosIndexString
```

```
'c' se encuentra en el indice 2
'a' se encuentra en el indice 13
'$' se encuentra en el indice -1

La ultima 'c' se encuentra en el indice 15
La ultima 'a' se encuentra en el indice 13
La ultima '$' se encuentra en el indice -1

"def" se encuentra en el indice 3
"def" se encuentra en el indice 16
"hola" se encuentra en el indice -1

La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "hola" se encuentra en el indice -1
```

Fig. 14.5 | Métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String` (parte 2 de 2).

Todas las búsquedas en este ejemplo se realizan en el objeto `String letras` (inicializado con "`abcdefghijklmabcdefghijklm`"). En las líneas 11 a 16 se utiliza el método `indexOf` para localizar la primera ocurrencia de un carácter en un objeto `String`. Si el método encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario devuelve `-1`. Hay dos versiones de `indexOf` que buscan caracteres en un objeto `String`. La expresión en la línea 12 utiliza la versión de `indexOf` que recibe una representación entera del carácter que se va a buscar. La expresión en la línea 14 utiliza otra versión del método `indexOf`, la cual recibe dos argumentos enteros: el carácter y el índice inicial en el que debe empezar la búsqueda en el objeto `String`.

En las líneas 19 a 24 se utiliza el método `lastIndexOf` para localizar la última ocurrencia de un carácter en un objeto `String`. El método realiza la búsqueda desde el final del objeto `String` hacia el inicio del mismo. Si encuentra el carácter, devuelve el índice de ese carácter en el objeto `String`; en caso contrario, devuelve `-1`. Hay dos versiones de `lastIndexOf` que buscan caracteres en un objeto `String`. La expresión en la línea 20 utiliza la versión que recibe la representación entera del carácter. La expresión en la línea 22 utiliza la versión que recibe dos argumentos enteros: la representación entera del carácter y el índice a partir del cual debe iniciarse la búsqueda *inversa* de ese carácter.

En las líneas 27 a 40 se demuestran las versiones de los métodos `indexOf` y `lastIndexOf` que reciben cada una de ellas un objeto `String` como el primer argumento. Estas versiones de los métodos se ejecutan en forma idéntica a las descritas anteriormente, excepto que buscan secuencias de caracteres (o subcadenas) que se especifican mediante sus argumentos `String`. Si se encuentra la subcadena, estos métodos devuelven el índice en el objeto `String` del primer carácter en la subcadena.

14.3.5 Extracción de subcadenas de las cadenas

La clase `String` proporciona dos métodos `substring` para permitir la creación de un nuevo objeto `String` al copiar parte de un objeto existente. Cada método devuelve un nuevo objeto `String`. Ambos métodos se muestran en la figura 14.6.

```
1  // Fig. 14.6: SubString.java
2  // Métodos substring de la clase String.
3
4  public class SubString
5  {
6     public static void main(String[] args)
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba los métodos substring
11       System.out.printf("La subcadena desde el indice 20 hasta el final es
            \"%s\"%n",
12          letras.substring(20) );
13       System.out.printf("%s \"%s\"%n",
14          "La subcadena desde el indice 3 hasta, pero sin incluir al 6 es",
15          letras.substring(3, 6) );
16    }
17 } // fin de la clase SubString
```

```
La subcadena desde el indice 20 hasta el final es "hijklm"
La subcadena desde el indice 3 hasta, pero sin incluir al 6 es "def"
```

Fig. 14.6 | Métodos `substring` de la clase `String`.

La expresión `letras.substring(20)` en la línea 12 utiliza el método `substring` que recibe un argumento entero. Este argumento especifica el índice inicial en el objeto `String` original `letras`, a partir del cual se van a copiar caracteres. La subcadena devuelta contiene una copia de los caracteres, desde el índice inicial hasta el final del objeto `String`. Si el índice especificado como argumento se encuentra fuera de los límites del objeto `String` se genera una excepción `StringIndexOutOfBoundsException`.

En la línea 15 se utiliza el método `substring` que recibe dos argumentos enteros: el índice inicial a partir del cual se van a copiar caracteres en el objeto `String` original y el índice que está una posición más allá del último carácter a copiar (es decir, que copia hasta pero *sin incluir a* ese índice en el objeto

String). La subcadena devuelta contiene copias de los caracteres especificados del objeto String original. Un índice que se encuentre fuera de los límites de la cadena genera una excepción StringIndexOutOfBoundsException.

14.3.6 Concatenación de cadenas

El método **concat** (figura 14.7) de String concatena dos objetos String (igual que cuando se usa el operador +) y devuelve un nuevo objeto String, el cual contiene los caracteres de ambos objetos String originales. La expresión s1.concat(s2) de la línea 13 forma un objeto String anexando los caracteres de la cadena s2 a los caracteres de la cadena s1. Los objetos String originales a los que se refieren s1 y s2 *no se modifican*.

```
1  // Fig. 14.7: ConcatenacionString.java
2  // Método concat de String.
3
4  public class ConcatenacionString
5  {
6     public static void main(String[] args)
7     {
8        String s1 = "Feliz ";
9        String s2 = "Cumpleanios";
10
11       System.out.printf("s1 = %s%ns2 = %s%n%n",s1, s2);
12       System.out.printf(
13          "Resultado de s1.concat(s2) = %s%n", s1.concat(s2));
14       System.out.printf("s1 despues de la concatenacion = %s%n", s1);
15    }
16 } // fin de la clase ConcatenacionString
```

```
s1 = Feliz
s2 = Cumpleanios

Resultado de s1.concat(s2) = Feliz Cumpleanios
s1 despues de la concatenacion= Feliz
```

Fig. 14.7 | Método concat de String.

14.3.7 Métodos varios de String

La clase String proporciona varios métodos que devuelven objetos String o arreglos de caracteres que contienen copias modificadas del contenido de un objeto String original, las cuales a su vez se modifican. Estos métodos (ninguno de los cuales modifica el objeto String sobre el cual se llaman) se demuestran en la figura 14.8.

```
1  // Fig. 14.8: VariosString2.java
2  // Métodos replace, toLowerCase, toUpperCase, trim y toCharArray de String.
3
4  public class VariosString2
5  {
6     public static void main(String[] args)
7     {
```

Fig. 14.8 | Métodos replace, toLowerCase, toUpperCase, trim y toCharArray de String (parte 1 de 2).

```
8         String s1 = "hola";
9         String s2 = "ADIOS";
10        String s3 = "   espacios   ";
11
12        System.out.printf("s1 = %s%ns2 = %s%ns3 = %s%n%n", s1, s2, s3);
13
14        // prueba del método replace
15        System.out.printf(
16           "Remplazar 'l' con 'L' en s1: %s%n%n", s1.replace('l', 'L'));
17
18        // prueba de toLowerCase y toUpperCase
19        System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
20        System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
21
22        // test trim method
23        System.out.printf("s3 despues de trim = \"%s\"%n%n", s3.trim());
24
25        // prueba del método toCharArray
26        char[] arregloChar = s1.toCharArray();
27        System.out.print("s1 como arreglo de caracteres = ");
28
29        for (char caracter : arregloChar)
30           System.out.print(caracter);
31
32        System.out.println();
33     }
34  } // fin de la clase VariosString2
```

```
s1 = hola
s2 = ADIOS
s3 =    espacios

Remplazar 'l' con 'L' en s1: hoLa

s1.toUpperCase() = HOLA
s2.toLowerCase() = adios

s3 despues de trim = "espacios"

s1 como arreglo de caracteres = hola
```

Fig. 14.8 | Métodos `replace`, `toLowerCase`, `toUpperCase`, `trim` y `toCharArray` de `String` (parte 2 de 2).

En la línea 16 se utiliza el método `replace` de `String` para devolver un nuevo objeto `String`, en el que cada ocurrencia del carácter 'l' (ele minúscula) en la cadena `s1` se reemplaza con el carácter 'L'. El método `replace` no modifica el objeto `String` original. Si no hay ocurrencias del primer argumento en el objeto `String`, el método `replace` devuelve el objeto `String` original. Una versión sobrecargada del método `replace` nos permite reemplazar subcadenas en vez de caracteres individuales.

En la línea 19 se utiliza el método **`toUpperCase`** de `String` para generar un nuevo objeto `String` con letras mayúsculas; sus correspondientes letras minúsculas existen en `s1`. El método devuelve un nuevo objeto `String` que contiene el objeto `String` convertido y deja el objeto `String` original sin cambios. Si no hay caracteres para convertir, el método `toUpperCase` devuelve el objeto `String` original.

En la línea 20 se utiliza el método **toLowerCase** de `String` para devolver un nuevo objeto `String` con letras minúsculas; sus correspondientes letras mayúsculas existen en `s2`. El objeto `String` original permanece sin cambios. Si no hay caracteres para convertir en el objeto `String` original, `toLowerCase` devuelve el objeto `String` original.

En la línea 23 se utiliza el método **trim** de `String` para generar un nuevo objeto `String` que elimine todos los caracteres de espacio en blanco que aparecen al principio o al final del objeto `String` en el que opera `trim`. El método devuelve un nuevo objeto `String` que contiene el objeto `String` sin espacios en blanco a la izquierda o a la derecha. El objeto `String` original permanece sin cambios. Si no hay caracteres de espacios en blanco al principio o al final, `trim` devuelve el objeto `String` original.

En la línea 26 se utiliza el método **toCharArray** de `String` para crear un nuevo arreglo de caracteres, el cual contiene una copia de los caracteres en la cadena `s1`. En las líneas 29 y 30 se imprime cada `char` en el arreglo.

14.3.8 Método valueOf de String

Como hemos visto, todo objeto en Java tiene un método `toString` que permite a un programa obtener la *representación de cadena* del objeto. Por desgracia, esta técnica no puede utilizarse con tipos primitivos, ya que éstos no tienen métodos. La clase `String` proporciona métodos `static` que reciben un argumento de cualquier tipo y lo convierten en un objeto `String`. En la figura 14.9 se demuestra el uso de los métodos **valueOf** de la clase `String`.

La expresión `String.valueOf(arregloChar)` en la línea 18 utiliza el arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. La expresión `String.valueOf(arregloChar, 3, 3)` de la línea 20 utiliza una porción del arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. El segundo argumento especifica el índice inicial a partir del cual se van a utilizar caracteres. El tercer argumento especifica el número de caracteres a usar.

```
1  // Fig. 14.9: StringValueOf.java
2  // Métodos valueOf de String.
3
4  public class StringValueOf
5  {
6     public static void main(String[] args)
7     {
8        char[] arregloChar = {'a', 'b', 'c', 'd', 'e', 'f'};
9        boolean valorBoolean = true;
10       char valorCaracter = 'Z';
11       int valorEntero = 7;
12       long valorLong = 10000000000L; // el sufijo L indica long
13       float valorFloat = 2.5f; // f indica que 2.5 es un float
14       double valorDouble = 33.333; // no hay sufijo, double es el predeterminado
15       Object refObjeto = "hola"; // asigna la cadena a una referencia Object
16
17       System.out.printf(
18          "arreglo de valores char = %s%n", String.valueOf(arregloChar));
19       System.out.printf("parte del arreglo char = %s%n",
20          String.valueOf(arregloChar, 3, 3));
21       System.out.printf(
22          "boolean = %s%n", String.valueOf(valorBoolean));
23       System.out.printf(
24          "char = %s%n", String.valueOf(valorCaracter));
```

Fig. 14.9 | Métodos `valueOf` de la clase `String` (parte 1 de 2).

```
25         System.out.printf("int = %s%n", String.valueOf(valorEntero));
26         System.out.printf("long = %s%n", String.valueOf(valorLong));
27         System.out.printf("float = %s%n", String.valueOf(valorFloat));
28         System.out.printf(
29            "double = %s%n", String.valueOf(valorDouble));
30         System.out.printf("Object = %s", String.valueOf(refObjeto));
31      }
32   } // fin de la clase StringValueOf
```

```
arreglo de valores char = abcdef
parte del arreglo char = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hola
```

Fig. 14.9 | Métodos `valueOf` de la clase `String` (parte 2 de 2).

Existen otras siete versiones del método `valueOf`, las cuales toman argumentos de tipo `boolean`, `char`, `int`, `long`, `float`, `double` y `Object`. Estas versiones se muestran en las líneas 21 a 30. La versión de `valueOf` que recibe un objeto `Object` como argumento puede hacerlo debido a que todos los objetos `Object` pueden convertirse en objetos `String` mediante el método `toString`.

[*Nota:* en las líneas 12 y 13 se utilizan los valores literales `10000000000L` y `2.5f` como valores iniciales de la variable `valorLong` tipo `long` y de la variable `valorFloat` tipo `float`, respectivamente. De manera predeterminada, Java trata a las literales enteras como tipo `int` y a las literales de punto flotante como tipo `double`. Si se anexa la letra `L` a la literal `10000000000` y se anexa la letra `f` a la literal `2.5`, se indica al compilador que `10000000000` debe tratarse como `long` y que `2.5` debe tratarse como `float`. Se puede usar una `L` mayúscula o una `l` minúscula para denotar una variable de tipo `long`, y una `F` mayúscula o una `f` minúscula para denotar una variable de tipo `float`].

14.4 La clase `StringBuilder`

Ahora hablaremos sobre las características de la clase **`StringBuilder`** para crear y manipular información de cadenas *dinámicas*; es decir, cadenas que pueden *modificarse*. Cada objeto `StringBuilder` es capaz de almacenar varios caracteres especificados por su *capacidad*. Si se excede la capacidad de un objeto `StringBuilder`, ésta se expande de manera automática para dar cabida a los caracteres adicionales.

Tip de rendimiento 14.3

Java puede realizar ciertas optimizaciones en las que se involucran objetos `String` (como hacer referencia a un objeto `String` desde múltiples variables), ya que sabe que estos objetos no cambiarán. Si los datos no van a cambiar, debemos usar objetos `String` (y no `StringBuilder`).

Tip de rendimiento 14.4

En los programas que realizan con frecuencia una concatenación de cadenas, u otras modificaciones de cadenas, a menudo es más eficiente implementar las modificaciones con la clase `StringBuilder`.

Observación de ingeniería de software 14.1

Los objetos `StringBuilder` no son seguros para usarse en hilos. Si varios hilos requieren acceso a la misma información de una cadena dinámica, use la clase `StringBuffer` en su código. Las clases `StringBuilder` y `StringBuffer` proveen capacidades idénticas, pero la clase `StringBuffer` es segura para los hilos. Si desea más información sobre trabajar con hilos, vea el capítulo 23 (en inglés en el sitio web del libro).

14.4.1 Constructores de `StringBuilder`

La clase `StringBuilder` proporciona cuatro constructores. En la figura 14.10 mostramos tres de ellos. En la línea 8 se utiliza el constructor de `StringBuilder` sin argumentos para crear un objeto `StringBuilder` que no contiene caracteres, y tiene una capacidad inicial de 16 caracteres (el valor predeterminado para un objeto `StringBuilder`). En la línea 9 se utiliza el constructor de `StringBuilder` que recibe un argumento entero para crear un objeto `StringBuilder` que no contiene caracteres, y su capacidad inicial se especifica mediante el argumento entero (es decir, 10). En la línea 10 se utiliza el constructor de `StringBuilder` que recibe un argumento `String` para crear un objeto `StringBuilder` que contiene los caracteres en el argumento `String`. La capacidad inicial es el número de caracteres en el argumento `String`, más 16.

En las líneas 12 a 14 se utiliza de manera implícita el método `toString` de la clase `StringBuilder` para imprimir los objetos `StringBuilder` con el método `printf`. En la sección 14.4.4, hablaremos acerca de cómo Java usa los objetos `StringBuilder` para implementar los operadores + y += para la concatenación de cadenas.

```
1  // Fig. 14.10: ConstructoresStringBuilder.java
2  // Constructores de StringBuilder.
3
4  public class ConstructoresStringBuilder
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder buffer1 = new StringBuilder();
9        StringBuilder buffer2 = new StringBuilder(10);
10       StringBuilder buffer3 = new StringBuilder("hola");
11
12       System.out.printf("buffer1 = \"%s\"%n", buffer1);
13       System.out.printf("buffer2 = \"%s\"%n", buffer2);
14       System.out.printf("buffer3 = \"%s\"%n", buffer3);
15    }
16 } // fin de la clase ConstructoresStringBuilder
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hola"
```

Fig. 14.10 | Constructores de la clase `StringBuilder`.

14.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder`

La clase `StringBuilder` proporciona los métodos **`length`** y **`capacity`** para devolver el número actual de caracteres en un objeto `StringBuilder` y el número de caracteres que pueden almacenarse en un objeto

StringBuilder sin necesidad de asignar más memoria, respectivamente. El método **ensureCapacity** garantiza que un StringBuilder tenga cuando menos la capacidad especificada. El método **setLength** incrementa o decrementa la longitud de un objeto StringBuilder. En la figura 14.11 se muestra el uso de estos métodos.

```
1  // Fig. 14.11: StringBuilderCapLen.java
2  // Métodos length, setLength, capacity y ensureCapacity de StringBuilder.
3
4  public class StringBuilderCapLen
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder buffer = new StringBuilder("Hola, como estas?");
9
10       System.out.printf("buffer = %s%nlongitud = %d%ncapacidad = %d%n%n",
11          buffer.toString(), buffer.length(), buffer.capacity());
12
13       buffer.ensureCapacity(75);
14       System.out.printf("Nueva capacidad = %d%n%n", buffer.capacity());
15
16       buffer.setLength(10));
17       System.out.printf("Nueva longitud = %d%nbuffer = %s%n",
18          buffer.length(), buffer.toString());
19    }
20 } // fin de la clase StringBuilderCapLen
```

```
bufer = Hola, como estas?
longitud = 17
capacidad = 33

Nueva capacidad = 75

Nueva longitud = 10
bufer = Hola, como
```

Fig. 14.11 | Métodos length, setLength, capacity y ensureCapacity de StringBuilder.

La aplicación contiene un objeto StringBuilder llamado bufer. En la línea 8 se utiliza el constructor de StringBuilder que toma un argumento String para inicializar el objeto StringBuilder con la cadena "Hola, como estas?". En las líneas 10 y 11 se imprime el contenido, la longitud y la capacidad del objeto StringBuilder. Observe en la ventana de salida que la capacidad inicial del objeto StringBuilder es de 33. Recuerde que el constructor de StringBuilder que recibe un argumento String inicializa la capacidad con la longitud de la cadena que se le pasa como argumento, más 16.

En la línea 13 se utiliza el método ensureCapacity para expandir la capacidad del objeto StringBuilder a un mínimo de 75 caracteres. En realidad, si la capacidad original es menor que el argumento, este método asegura una capacidad que sea el valor mayor entre el número especificado como argumento y el doble de la capacidad original más 2. La capacidad del objeto StringBuilder permanece sin cambios si es mayor que la capacidad especificada.

Tip de rendimiento 14.5

El proceso de incrementar en forma dinámica la capacidad de un objeto `StringBuilder` *puede requerir una cantidad considerable de tiempo. La ejecución de un gran número de estas operaciones puede degradar el rendimiento de una aplicación. Si un objeto* `StringBuilder` *va a aumentar su tamaño en forma considerable (quizás varias veces), establecer desde el principio su capacidad a un nivel alto incrementará el rendimiento.*

En la línea 16 se utiliza el método `setLength` para establecer la longitud del objeto `StringBuilder` en 10. Si la longitud especificada es menor que el número actual de caracteres en el objeto `StringBuilder`, el búfer se trunca a la longitud especificada (es decir, los caracteres en el objeto `StringBuilder` que excedan la longitud especificada serán descartados). Si la longitud especificada es mayor que el número de caracteres que hay actualmente en el objeto `StringBuilder`, se anexan caracteres `null` (caracteres con la representación numérica de 0) hasta que el número total de caracteres en el objeto `StringBuilder` sea igual a la longitud especificada.

14.4.3 Métodos charAt, setCharAt, getChars y reverse de StringBuilder

La clase `StringBuilder` proporciona los métodos **`charAt`**, **`setCharAt`**, **`getChars`** y **`reverse`** para manipular los caracteres en un objeto `StringBuilder` (figura 14.12). El método `charAt` (línea 12) recibe un argumento entero y devuelve el carácter que se encuentre en el objeto `StringBuilder` en ese índice. El método `getChars` (línea 15) copia los caracteres de un objeto `StringBuilder` al arreglo de caracteres que recibe como argumento. Este método recibe cuatro argumentos: el índice inicial a partir del cual deben copiarse caracteres en el objeto `StringBuilder`, el índice una posición más allá del último carácter a copiar del objeto `StringBuilder`, el arreglo de caracteres en el que se van a copiar los caracteres y la posición inicial en el arreglo de caracteres en donde debe colocarse el primer carácter. El método `setCharAt` (líneas 21 y 22) recibe un entero y un carácter como argumentos y asigna el carácter en la posición especificada en el objeto `StringBuilder` al carácter que recibe como argumento. El método `reverse` (línea 25) invierte el contenido del objeto `StringBuilder`. Al tratar de acceder a un carácter que se encuentra fuera de los límites de un objeto `StringBuilder` se produce una excepción `StringIndexOutOfBoundsException`.

```
1  // Fig. 14.12: StringBuilderChars.java
2  // Métodos charAt, setCharAt, getChars y reverse de StringBuilder.
3
4  public class StringBuilderChars
5  {
6     public static void main(String[] args)
7     {
8        StringBuilder bufer = new StringBuilder("hola a todos");
9
10       System.out.printf("bufer = %s%n", bufer.toString());
11       System.out.printf("Caracter en 0: %s%nCaracter en 3: %s%n%n",
12          bufer.charAt(0), bufer.charAt(4));
13
14       char[] arregloChars = new char[bufer.length()];
15       bufer.getChars(0, bufer.length(), arregloChars, 0);
16       System.out.print("Los caracteres son: ");
```

Fig. 14.12 | Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` (parte 1 de 2).

```
17
18          for (char caracter : arregloChars)
19             System.out.print(caracter);
20
21          bufer.setCharAt(0, 'H');
22          bufer.setCharAt(6, 'T');
23          System.out.printf("%n%nbufer = %s", bufer.toString());
24
25          bufer.reverse();
26          System.out.printf("%n%nbufer = %s%n", bufer.toString());
27       }
28    } // fin de la clase StringBuilderChars
```

```
bufer  = hola a todos
Caracter en 0: h
Caracter en 3: a

Los caracteres son: hola a todos

bufer  = Hola a Todos

bufer  = sodoT a aloH
```

Fig. 14.12 | Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` (parte 2 de 2).

14.4.4 Métodos append de StringBuilder

La clase `StringBuilder` proporciona métodos **append** *sobrecargados* (figura 14.13) para permitir que se agreguen valores de diversos tipos al final de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos y para arreglos de caracteres, objetos `String`, `Object` y demás (recuerde que el método `toString` produce una representación de cadena de cualquier objeto `Object`). Cada método recibe su argumento, lo convierte en una cadena y lo anexa al objeto `StringBuilder`.

```
1  // Fig. 14.13: StringBuilderAppend.java
2  // Métodos append de StringBuilder.
3
4  public class StringBuilderAppend
5  {
6     public static void main(String[] args)
7     {
8        Object refObjeto = "hola";
9        String cadena = "adios";
10       char[] arregloChar = {'a', 'b', 'c', 'd', 'e', 'f'};
11       boolean valorBoolean = true;
12       char valorChar = 'Z';
13       int valorInt = 7;
14       long valorLong = 10000000000L;
15       float valorFloat = 2.5f;
16       double valorDouble = 33.333;
```

Fig. 14.13 | Métodos `append` de la clase `StringBuilder` (parte 1 de 2).

```
17
18          StringBuilder ultimoBufer = new StringBuilder("ultimo bufer");
19          StringBuilder bufer = new StringBuilder();
20
21          bufer.append(refObjeto)
22              .append("%n")
23              .append(cadena)
24              .append("%n")
25              .append(arregloChar)
26              .append("%n")
27              .append(arregloChar, 0, 3)
28              .append("%n")
29              .append(valorBoolean)
30              .append("%n")
31              .append(valorChar);
32              .append("%n")
33              .append(valorInt)
34              .append("%n")
35              .append(valorLong)
36              .append("%n")
37              .append(valorFloat)
38              .append("%n")
39              .append(valorDouble)
40              .append("%n")
41              .append(ultimoBufer);
42
43          System.out.printf("bufer contiene%n%s%n", bufer.toString());
44      }
45  } // fin de StringBuilderAppend
```

```
bufer contiene
hola
adios
abcdef
abc
true
Z
7
10000000000
2.5
33.333
ultimo bufer
```

Fig. 14.13 | Métodos `append` de la clase `StringBuilder` (parte 2 de 2).

El compilador puede usar `StringBuilder` y los métodos `append` para implementar los operadores + y += de concatenación `String`. Por ejemplo, suponga que se realizan las siguientes declaraciones:

```
String cadena1 = "hola";
String cadena2 = "BC";
int valor = 22;
```

la instrucción

```
String s = cadena1 + cadena2 + valor;
```

concatena a "hola", "BC" y 22. La concatenación se puede realizar de la siguiente manera:

```
String s = new StringBuilder().append("hola").append("BC").
   append(22).toString();
```

Primero, la instrucción anterior crea un objeto `StringBuilder` *vacío* y después le anexa las cadenas "hola", "BC" y el entero 22. A continuación, el método `toString` de `StringBuilder` convierte el objeto `StringBuilder` en un objeto `String` que se asigna al objeto `String s`. La instrucción

```
s += "!";
```

se ejecuta de la siguiente manera (esto puede variar de un compilador a otro):

```
s = new StringBuilder().append(s).append("!").toString();
```

Esto crea un objeto `StringBuilder` vacío y después se le anexa el contenido actual de `s`, seguido por "!". A continuación, el método `toString` de `StringBuilder` (que debe llamarse de manera *explícita* aquí) devuelve el contenido del objeto `StringBuilder` como un objeto `String`, y el resultado se asigna a `s`.

14.4.5 Métodos de inserción y eliminación de StringBuilder

La clase `StringBuilder` proporciona métodos **`insert`** sobrecargados para permitir que se inserten valores de diversos tipos en cualquier posición de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos, y para arreglos de caracteres, objetos `String`, `Object` y `CharSequence`. Cada uno de los métodos toma su segundo argumento y lo inserta en el índice especificado por el primer argumento. Si el primer argumento es menor que 0 o mayor que la longitud del objeto `StringBuilder`, ocurre una excepción `StringIndexOutOfBoundsException`. La clase `StringBuilder` también proporciona los métodos **`delete`** y **`deleteCharAt`** para eliminar caracteres en cualquier posición de un objeto `StringBuilder`. El método `delete` recibe dos argumentos: el índice inicial y el índice que se encuentra una posición más allá del último de los caracteres que se van a eliminar. Se eliminan todos los caracteres que empiezan en el índice inicial hasta, pero *sin* incluir al índice final. El método `deleteCharAt` recibe un argumento que es el índice del carácter a eliminar. El uso de índices inválidos hace que ambos métodos lancen una excepción `StringIndexOutOfBoundsException`. En la figura 14.14 se muestran los métodos `insert`, `delete` y `deleteCharAt`.

```
1  // Fig. 14.14: StringBuilderInsertDelete.java
2  // Métodos insert, delete y deleteCharAt de StringBuilder.
3
4  public class StringBuilderInsertDelete
5  {
6     public static void main(String[] args)
7     {
8        Object refObjeto = "hola";
9        String cadena = "adios";
10       char[] arregloChars = {'a', 'b', 'c', 'd', 'e', 'f'};
11       boolean valorBoolean = true;
12       char valorChar = 'K';
13       int valorInt = 7;
14       long valorLong = 10000000;
15       float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
16       double valorDouble = 33.333;
17
```

Fig. 14.14 | Métodos `insert`, `delete` y `deleteCharAt` de `StringBuilder` (parte 1 de 2).

```
18         StringBuilder bufer = new StringBuilder();
19
20         bufer.insert(0, refObjeto);
21         bufer.insert(0, "  "); // cada uno de estos contiene dos espacios
22         bufer.insert(0, cadena);
23         bufer.insert(0, "  ");
24         bufer.insert(0, arregloChars);
25         bufer.insert(0, "  ");
26         bufer.insert(0, arregloChars, 3, 3);
27         bufer.insert(0, "  ");
28         bufer.insert(0, valorBoolean);
29         bufer.insert(0, "  ");
30         bufer.insert(0, valorChar);
31         bufer.insert(0, "  ");
32         bufer.insert(0, valorInt);
33         bufer.insert(0, "  ");
34         bufer.insert(0, valorLong);
35         bufer.insert(0, "  ");
36         bufer.insert(0, valorFloat);
37         bufer.insert(0, "  ");
38         bufer.insert(0, valorDouble);
39
40         System.out.printf(
41            "bufer despues de insertar:%n%s%n%n", bufer.toString());
42
43         bufer.deleteCharAt(10); // elimina el 5 en 2.5
44         bufer.delete(2, 6); // elimina el .333 en 33.333
45
46         System.out.printf(
47            "bufer despues de eliminar:%n%s%n", bufer.toString());
48      }
49   } // fin de la clase StringBuilderInsertDelete
```

```
bufer despues de insertar:
33.333  2.5  10000000  7  K  true  def  abcdef  adios  hola

bufer despues de eliminar:
33  2.  10000000  7  K  true  def  abcdef  adios  hola
```

Fig. 14.14 | Métodos `insert`, `delete` y `deleteCharAt` de `StringBuilder` (parte 2 de 2).

14.5 La clase Character

Java proporciona ocho **clases de envoltura de tipos** (`Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` y `Long`) las cuales permiten que los valores de tipo primitivo se traten como objetos. En esta sección presentaremos la clase `Character` que es la clase de envoltura de tipos para el tipo primitivo `char`.

La mayoría de los métodos de la clase `Character` son `static`, diseñados para facilitar el procesamiento de valores `char` individuales. Estos métodos reciben cuando menos un argumento tipo carácter y realizan una prueba o una manipulación del carácter. Esta clase también contiene un constructor que recibe un argumento `char` para inicializar un objeto `Character`. En los siguientes tres ejemplos presentaremos la mayor

parte de los métodos de la clase `Character`. Para obtener más información sobre esta clase (y las demás clases de envoltura de tipos), consulte el paquete `java.lang` en la documentación de la API de Java.

En la figura 14.15 se muestran algunos métodos `static` que prueban caracteres para determinar si son de un tipo de carácter específico y los métodos `static` que realizan conversiones de caracteres de minúscula a mayúscula, y viceversa. Puede introducir cualquier carácter y aplicar estos métodos a ese carácter.

```
1  // Fig. 14.15: MetodosStaticChar.java
2  // Los métodos static de Character para probar caracteres y conversión de
   mayúsculas/minúsculas.
3  import java.util.Scanner;
4
5  public class MetodosStaticChar
6  {
7     public static void main(String[] args)
8     {
9        Scanner scanner = new Scanner(System.in); // crea objeto scanner
10       System.out.println("Escriba un caracter y oprima Intro");
11       String entrada = scanner.next();
12       char c = entrada.charAt(0); // obtiene el caracter de entrada
13
14       // muestra información sobre los caracteres
15       System.out.printf("esta definido: %b%n", Character.isDefined(c));
16       System.out.printf("es digito: %b%n", Character.isDigit(c));
17       System.out.printf("es el primer caracter en un identificador de Java: %b%n",
18          Character.isJavaIdentifierStart(c));
19       System.out.printf("es parte de un identificador de Java: %b%n",
20          Character.isJavaIdentifierPart(c));
21       System.out.printf("es letra: %b%n", Character.isLetter(c));
22       System.out.printf(
23          "es letra o digito: %b%n", Character.isLetterOrDigit(c));
24       System.out.printf(
25          "es minuscula: %b%n", Character.isLowerCase(c));
26       System.out.printf(
27          "es mayuscula: %b%n", Character.isUpperCase(c));
28       System.out.printf(
29          "a mayuscula: %s%n", Character.toUpperCase(c));
30       System.out.printf(
31          "a minuscula: %s%n", Character.toLowerCase(c));
32    }
33 } // fin de la clase MetodosStaticChar
```

```
Escriba un caracter y oprima Intro
A
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: true
es letra o digito: true
es minuscula: false
es mayuscula: true
a mayuscula: A
a minuscula: a
```

Fig. 14.15 | Métodos `static` de `Character` para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa (parte 1 de 2).

```
Escriba un caracter y oprima Intro
8
esta definido: true
es digito: true
es el primer caracter en un identificador de Java: false
es parte de un identificador de Java: true
es letra: false
es letra o digito: true
es minuscula: false
es mayuscula: false
a mayuscula: 8
a minuscula: 8
```

```
Escriba un caracter y oprima Intro
$
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: false
es letra o digito: false
es minuscula: false
es mayuscula: false
a mayuscula: $
a minuscula: $
```

Fig. 14.15 | Métodos `static` de `Character` para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa (parte 2 de 2).

En la línea 15 se utiliza el método **`isDefined`** de `Character` para determinar si el carácter `c` está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve `true`; en caso contrario, devuelve `false`. En la línea 16 se utiliza el método **`isDigit`** de `Character` para determinar si el carácter `c` es un dígito definido en Unicode. De ser así el método devuelve `true` y, en caso contrario devuelve `false`.

En la línea 18 se utiliza el método **`isJavaIdentifierStart`** de `Character` para determinar si `c` es un carácter que puede ser el primer carácter de un identificador en Java; es decir, una letra, un guion bajo (_) o un signo de dólares ($). De ser así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 20 se utiliza el método **`isJavaIdentifierPart`** de `Character` para determinar si el carácter `c` puede utilizarse en un identificador en Java; es decir, un dígito, una letra, un guion bajo (_) o un signo de dólares ($). De ser así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 21 se utiliza el método **`isLetter`** de `Character` para determinar si el carácter `c` es una letra. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 23 se utiliza el método **`isLetterOrDigit`** de `Character` para determinar si el carácter `c` es una letra o un dígito. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 25 se utiliza el método **`isLowerCase`** de `Character` para determinar si el carácter `c` es una letra minúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 27 se utiliza el método **`isUpperCase`** de `Character` para determinar si el carácter `c` es una letra mayúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 29 se utiliza el método **`toUpperCase`** de `Character` para convertir el carácter `c` en su letra mayúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en

mayúscula; en caso contrario, el método devuelve su argumento original. En la línea 31 se utiliza el método **toLowerCase** de Character para convertir el carácter c en su letra minúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en minúscula; en caso contrario, el método devuelve su argumento original.

En la figura 14.16 se muestran los métodos static **digit** y **forDigit** de Character, los cuales convierten caracteres a dígitos y dígitos a caracteres, respectivamente, en distintos sistemas numéricos. Los sistemas numéricos comunes son el decimal (base 10), octal (base 8), hexadecimal (base 16) y binario (base 2). La base de un número se conoce también como su **raíz**. Para obtener más información sobre las conversiones entre sistemas numéricos, vea el apéndice J.

```
1  // Fig. 14.16: MetodosStaticChar2.java
2  // Métodos de conversión estáticos de la clase Character.
3  import java.util.Scanner;
4
5  public class MetodosStaticChar2
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       Scanner scanner = new Scanner(System.in);
11
12       // obtiene la raíz
13       System.out.println("Escriba una raiz:");
14       int raiz = scanner.nextInt();
15
16       // obtiene la selección del usuario
17       System.out.printf("Seleccione una opcion:%n1 -- %s%n2 -- %s%n",
18          "Convertir digito a caracter", "Convertir caracter a digito");
19       int opcion = scanner.nextInt();
20
21       // procesa la petición
22       switch (opcion)
23       {
24          case 1: // convierte dígito a carácter
25             System.out.println("Escriba un digito:");
26             int digito = scanner.nextInt();
27             System.out.printf("Convertir digito a caracter: %s%n",
28                Character.forDigit(digito, raiz));
29             break;
30
31          case 2: // convierte carácter a dígito
32             System.out.println("Escriba un caracter:");
33             char caracter = scanner.next().charAt(0);
34             System.out.printf("Convertir caracter a digito: %s%n",
35                Character.digit(caracter, raiz));
36             break;
37       }
38    }
39 } // fin de la clase MetodosStaticChar2
```

Fig. 14.16 | Métodos de conversión static de la clase Character (parte 1 de 2).

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
2
Escriba un caracter:
A
Convertir caracter a digito: 10
```

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
1
Escriba un digito:
13
Convertir digito a caracter: d
```

Fig. 14.16 | Métodos de conversión `static` de la clase `Character` (parte 2 de 2).

En la línea 28 se utiliza el método `forDigit` para convertir el entero `digito` en un carácter del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el entero decimal `13` en base 16 (la `raiz`) tiene el valor de carácter '`d`'. Las letras en minúsculas y mayúsculas representan el *mismo* valor en los sistemas numéricos. En la línea 35 se utiliza el método `digit` para convertir la variable `caracter` en un entero del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el carácter '`A`' es la representación en base 16 (la `raiz`) del valor 10 en base 10. La raíz debe estar entre 2 y 36, inclusive.

En la figura 14.17 se muestra el constructor y varios métodos de instancia de la clase `Character`, como **`charValue`**, `toString` y `equals`. En las líneas 7 y 8 se instancian dos objetos `Character` al asignar las constantes de caracteres '`A`' y '`a`', respectivamente, a las variables `Character`. Java convierte de manera automática estas literales `char` en objetos `Character`: un proceso conocido como *autoboxing*, que veremos con más detalle en la sección 16.4. En la línea 11 se utiliza el método `charValue` de `Character` para devolver el valor `char` almacenado en el objeto `Character` llamado `c1`. En la línea 11 se devuelve la representación de cadena del objeto `Character` llamado `c2`, utilizando el método `toString`. La condición en la línea 13 utiliza el método `equals` para determinar si el objeto `c1` tiene el mismo contenido que el objeto `c2` (es decir, si los caracteres dentro de cada objeto son iguales).

```
1  // Fig. 14.17: OtrosMetodosChar.java
2  // Métodos no static de la clase Character.
3  public class OtrosMetodosChar
4  {
5     public static void main(String[] args)
6     {
7        Character c1 = 'A';
8        Character c2 = 'a';
```

Fig. 14.17 | Métodos de instancia de la clase `Character` (parte 1 de 2).

```
9
10          System.out.printf(
11             "c1 = %s%nc2 = %s%n%n", c1.charValue(), c2.toString());
12
13          if (c1.equals(c2))
14             System.out.println("c1 y c2 son iguales%n");
15          else
16             System.out.println("c1 y c2 no son iguales%n");
17       }
18    } // fin de la clase OtrosMetodosChar
```

```
c1 = A
c2 = a

c1 y c2 no son iguales
```

Fig. 14.17 | Métodos de instancia de la clase `Character` (parte 2 de 2).

14.6 División de objetos `String` en tokens

Cuando usted lee una oración, su mente la divide en señales (palabras individuales y signos de puntuación, cada uno de los cuales transfiere a usted su significado). Los compiladores también dividen objetos en señales, llamadas **tokens**. Los tokens descomponen instrucciones en piezas individuales tales como palabras clave, identificadores, operadores y demás elementos de un lenguaje de programación. Ahora estudiaremos el método **`split`** de la clase `String` que descompone un objeto `String` en los tokens que lo componen. Los tokens se separan unos de otros mediante **delimitadores**, que por lo general son caracteres de espacio en blanco tales como espacios, tabuladores, nuevas líneas y retornos de carro. También pueden utilizarse otros caracteres como delimitadores para separar tokens. La aplicación de la figura 14.18 muestra el uso del método `split` de `String`.

Cuando el usuario oprime *Intro*, el enunciado de entrada se almacena en la variable `enunciado`. La línea 17 invoca el método `split` de `String` con el argumento " ", que devuelve un arreglo de objetos `String`. El carácter de espacio en el argumento `String` es el delimitador que el método `split` utiliza para localizar los tokens en el objeto `String`. Como veremos en la siguiente sección, el argumento para el método `split` puede ser una expresión regular para realizar descomposiciones en tokens más complejas. En la línea 19 se muestra en pantalla la longitud del arreglo `tokens`; es decir, el número de tokens en `enunciado`. En las líneas 21 y 22 se imprime cada token en una línea independiente.

```
1  // Fig. 14.18: PruebaToken.java
2  // Uso de un objeto StringTokenizer para descomponer objetos String en tokens.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class PruebaToken
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // obtiene el enunciado
12       Scanner scanner = new Scanner(System.in);
```

Fig. 14.18 | Objeto `StringTokenizer` utilizado para descomponer cadenas en tokens (parte 1 de 2).

```
13          System.out.println("Escriba un enunciado y oprima Intro");
14          String enunciado = scanner.nextLine();
15
16          // procesa el enunciado del usuario
17          String[] tokens = enunciado.split(" ");
18          System.out.printf("Numero de elementos: %d%nLos tokens son:%n",
19             tokens.length);
20
21          for (String token : tokens)
22            System.out.println(token);
23       }
24    } // fin de la clase PruebaToken
```

```
Escriba un enunciado y oprima Intro
Este es un enunciado con siete tokens
Numero de elementos: 7
Los tokens son:
Este
es
un
enunciado
con
siete
tokens
```

Fig. 14.18 | Objeto `StringTokenizer` utilizado para descomponer cadenas en tokens (parte 2 de 2).

14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

Una **expresión regular** es un objeto `String` que describe un *patrón de búsqueda* para *relacionar* caracteres en otros objetos `String`. Dichas expresiones son útiles para *validar la entrada* y asegurar que los datos estén en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y un apellido sólo debe contener letras, espacios, apóstrofes y guiones cortos. Una aplicación de las expresiones regulares es facilitar la construcción de un compilador. A menudo se utiliza una expresión regular larga y compleja para *validar la sintaxis de un programa*. Si el código del programa *no* coincide con la expresión regular, el compilador sabe que hay un error de sintaxis dentro del código.

La clase `String` proporciona varios métodos para realizar operaciones con expresiones regulares, siendo la más simple la operación de concordancia. El método **`matches`** de la clase `String` recibe un objeto `String` que especifica la expresión regular, e iguala el contenido del objeto `String` que lo llama con la expresión regular. Este método devuelve un valor de tipo `boolean`, indicando si hubo concordancia o no.

Una expresión regular consiste de caracteres literales y símbolos especiales. La figura 14.19 especifica algunas **clases predefinidas de caracteres** que pueden usarse con las expresiones regulares. Una clase de carácter es una *secuencia de escape* que representa a un grupo de caracteres. Un dígito es cualquier carácter numérico. Un **carácter de palabra** es cualquier letra (mayúscula o minúscula), cualquier dígito o el carácter de guion bajo. Un carácter de espacio en blanco es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Cada clase de carácter se iguala con un solo carácter en el objeto `String` que intentamos hacer concordar con la expresión regular.

Carácter	Concuerda con	Carácter	Concuerda con
\d	cualquier dígito	\D	cualquier carácter que no sea dígito
\w	cualquier carácter de palabra	\W	cualquier carácter que no sea de palabra
\s	cualquier espacio en blanco	\S	cualquier carácter que no sea de espacio en blanco

Fig. 14.19 | Clases predefinidas de caracteres.

Las expresiones regulares no están limitadas a esas clases predefinidas de caracteres. Las expresiones utilizan varios operadores y otras formas de notación para igualar patrones complejos. Analizaremos varias de estas técnicas en la aplicación de las figuras 14.20 y 14.21, la cual *valida la entrada del usuario* mediante expresiones regulares. [*Nota:* esta aplicación no está diseñada para igualar todos los posibles datos de entrada del usuario].

```
1  // Fig. 14.20: ValidacionEntrada.java
2  // Valida la información del usuario mediante expresiones regulares.
3
4  public class ValidacionEntrada
5  {
6     // valida el primer nombre
7     public static boolean validarPrimerNombre(String primerNombre)
8     {
9        return primerNombre.matches("[A-Z][a-zA-Z]*");
10    }
11
12    // valida el apellido
13    public static boolean validarApellidoPaterno(String apellidoPaterno)
14    {
15       return apellidoPaterno.matches("[a-zA-z]+(['-][a-zA-Z]+)*");
16    }
17
18    // valida la dirección
19    public static boolean validarDireccion(String direccion)
20    {
21       return direccion.matches(
22          "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
23    }
24
25    // valida la ciudad
26    public static boolean validarCiudad(String ciudad)
27    {
28       return ciudad.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
29    }
30
31    // valida el estado
32    public static boolean validarEstado(String estado)
33    {
```

Fig. 14.20 | Validación de la información del usuario mediante expresiones regulares (parte 1 de 2).

```
34          return estado.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
35       }
36
37       // valida el código postal
38       public static boolean validarCP(String cp)
39       {
40          return zip.matches("\\d{5}");
41       }
42
43       // valida el teléfono
44       public static boolean validarTelefono(String telefono)
45       {
46          return telefono.matches("[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}");
47       }
48    } // fin de la clase ValidacionEntrada
```

Fig. 14.20 | Validación de la información del usuario mediante expresiones regulares (parte 2 de 2).

```
1  // Fig. 14.21: Validacion.java
2  // Recibe y valida la información del usuario mediante la clase ValidacionEntrada.
3  import java.util.Scanner;
4
5  public class Validacion
6  {
7     public static void main(String[] args)
8     {
9        // obtiene la entrada del usuario
10       Scanner scanner = new Scanner(System.in);
11       System.out.println("Escriba el primer nombre:");
12       String primerNombre = scanner.nextLine();
13       System.out.println("Escriba el apellido paterno:");
14       String apellidoPaterno = scanner.nextLine();
15       System.out.println("Escriba la direccion:");
16       String direccion = scanner.nextLine();
17       System.out.println("Escriba la ciudad:");
18       String ciudad = scanner.nextLine();
19       System.out.println("Escriba el estado:");
20       String estado = scanner.nextLine();
21       System.out.println("Escriba el codigo postal:");
22       String cp = scanner.nextLine();
23       System.out.println("Escriba el telefono:");
24       String telefono = scanner.nextLine();
25
26       // valida la entrada del usuario y muestra mensaje de error
27       System.out.println("%nValidar resultado:");
28
29       if (!ValidacionEntrada.validarPrimerNombre(primerNombre))
30          System.out.println("Primer nombre invalido");
31       else if (!ValidacionEntrada.validarApellidoPaterno(apellidoPaterno))
32          System.out.println("Apellido paterno invalido");
33       else if (!ValidacionEntrada.validarDireccion(direccion))
34          System.out.println("Direccion invalida");
```

Fig. 14.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada` (parte 1 de 2).

```
35         else if (!ValidacionEntrada.validarCiudad(ciudad))
36            System.out.println("Ciudad invalida");
37         else if (!ValidacionEntrada.validarEstado(estado))
38            System.out.println("Estado invalido");
39         else if (!ValidacionEntrada.validarCP(cp))
40            System.out.println("Codigo postal invalido");
41         else if (!ValidacionEntrada.validarTelefono(telefono))
42            System.out.println("Numero telefonico invalido");
43         else
44            System.out.println("La entrada es valida.  Gracias.");
45      }
46   } // fin de la clase Validacion
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Cierta Calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
123
Escriba el telefono:
123-456-7890

Validar resultado:
Codigo postal invalido
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Una calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
12345
Escriba el telefono:
123-456-7890

Validar resultado:
La entrada es valida.  Gracias.
```

Fig. 14.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada` (parte 2 de 2).

En la figura 14.20 se valida la entrada del usuario. En la línea 9 se valida el nombre. Para hacer que concuerde un conjunto de caracteres que no tiene una clase predefinida de carácter, utilice los corchetes

([]). Por ejemplo, el patrón "[aeiou]" puede utilizarse para concordar con una sola vocal. Los rangos de caracteres pueden representarse colocando un guion corto (-) entre dos caracteres. En el ejemplo, "[A-Z]" concuerda con una sola letra mayúscula. Si el primer carácter entre corchetes es "^", la expresión acepta cualquier carácter distinto a los que se indiquen. Sin embargo, es importante observar que "[^Z]" no es lo mismo que "[A-Y]", la cual concuerda con las letras mayúsculas A-Y; "[^Z]" concuerda con *cualquier carácter distinto de* la letra Z mayúscula, incluyendo las letras minúsculas y los caracteres que no son letras, como el carácter de nueva línea. Los rangos en las clases de caracteres se determinan mediante los valores enteros de las letras. En este ejemplo, "[A-Za-z]" concuerda con todas las letras mayúsculas y minúsculas. El rango "[A-z]" concuerda con todas las letras y también concuerda con los caracteres (como [y \) que tengan un valor entero entre la letra Z mayúscula y la letra a minúscula (para obtener más información acerca de los valores enteros de los caracteres, consulte el apéndice B). Al igual que las clases predefinidas de caracteres, las clases de caracteres delimitadas entre corchetes concuerdan con un solo carácter en el objeto de búsqueda.

En la línea 9, el asterisco después de la segunda clase de carácter indica que puede concordar cualquier número de letras. En general, cuando aparece el operador de expresión regular "*" en una expresión regular, la aplicación intenta hacer que concuerden cero o más ocurrencias de la subexpresión que va inmediatamente después de "*". El operador "+" intenta hacer que concuerden una o más ocurrencias de la subexpresión que va inmediatamente después de "+". Por lo tanto, "A*" y "A+" concordarán con "AAA" o "A", pero sólo "A*" concordará con una cadena vacía.

Si el método validarPrimerNombre devuelve true (línea 29 de la figura 14.21), la aplicación trata de validar el apellido paterno (línea 31) llamando a validarApellidoPaterno (líneas 13 a 16 de la figura 14.20). La expresión regular para validar el apellido concuerda con cualquier número de letras divididas por espacios, apóstrofes o guiones cortos.

En la línea 33 de la figura 14.21 se hace una llamada al método validarDireccion (líneas 19 a 23 de la figura 14.20) para validar la dirección. La primera clase de carácter concuerda con cualquier dígito una o más veces (\\d+). Se utilizan dos caracteres \, ya que el carácter \ por lo general inicia una secuencia de escape en una cadena. Por lo tanto, \\d en un objeto String representa al patrón de expresión regular \d. Después concordamos uno o más caracteres de espacio en blanco (\\s+). El carácter "|" concuerda con la expresión a su izquierda o a su derecha. Por ejemplo, "Hola (Juan|Juana)" concuerda tanto con "Hola Juan" como con "Hola Juana". Los paréntesis se utilizan para agrupar partes de la expresión regular. En este ejemplo, el lado izquierdo de | concuerda con una sola palabra y el lado derecho concuerda con dos palabras separadas por cualquier cantidad de espacios en blanco. Por lo tanto, la dirección debe contener un número seguido de una o dos palabras. Así, "10 Broadway" y "10 Main Street" son ambas direcciones válidas en este ejemplo. Los métodos ciudad (líneas 26 a 29 de la figura 14.20) y estado (líneas 32 a 35 de la figura 14.20) también concuerdan con cualquier palabra que tenga al menos un carácter o, de manera alternativa, con dos palabras cualesquiera con al menos un carácter, si éstas van separadas por un solo espacio. Esto significa que tanto Waltham como West Newton concordarían.

Cuantificadores

El asterisco (*) y el signo de suma (+) se conocen de manera formal como **cuantificadores**. En la figura 14.22 se presentan todos los cuantificadores. Ya hemos visto cómo funcionan el asterisco (*) y el signo de suma (+). Todos los cuantificadores afectan solamente a la subexpresión que va inmediatamente antes del cuantificador. El cuantificador signo de interrogación (?) concuerda con cero o una ocurrencia de la expresión que cuantifica. Un conjunto de llaves que contienen un número ({n}) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. En la línea 40 de la figura 14.20 mostramos este cuantificador para validar el código postal. Si se incluye una coma después del número encerrado entre llaves, el cuantificador concordará al menos con *n* ocurrencias de la expresión cuantificada. El conjunto de llaves que contienen dos números ({*n*,*m*}) concuerda entre *n* y *m* ocurrencias de la expresión que califica. Los cuantificadores pueden aplicarse a patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Cuantificador	Concuerda con
*	Concuerda con cero o más ocurrencias del patrón.
+	Concuerda con una o más ocurrencias del patrón.
?	Concuerda con cero o una ocurrencia del patrón.
{*n*}	Concuerda con exactamente *n* ocurrencias.
{*n*,}	Concuerda con al menos *n* ocurrencias.
{*n*,*m*}	Concuerda con entre *n* y *m* (inclusive) ocurrencias.

Fig. 14.22 | Cuantificadores utilizados en expresiones regulares.

Se dice que todos los cuantificadores son **avaros**. Esto significa que concordarán con todas las ocurrencias que puedan, siempre y cuando haya concordancia. No obstante, si alguno de estos cuantificadores va seguido por un signo de interrogación (?), el cuantificador se vuelve **reacio** (en ocasiones llamado **flojo**). De esta forma, concordará con la menor cantidad de ocurrencias posibles, siempre y cuando haya concordancia.

El código postal (línea 40 en la figura 14.20) concuerda con un dígito cinco veces. Esta expresión regular utiliza la clase de carácter de dígito y un cuantificador con el dígito 5 entre llaves. El número telefónico (línea 46 en la figura 14.20) concuerda primero con tres dígitos (el primero no puede ser cero) seguidos de un guion corto, seguido a su vez de tres dígitos más (de nuevo, el primero no puede ser cero), seguidos finalmente de cuatro dígitos más.

El método `matches` de `String` verifica si una cadena completa se conforma a una expresión regular. Por ejemplo, queremos aceptar "`Smith`" como apellido, pero no "`9@Smith#`". Si sólo una subcadena concuerda con la expresión regular, el método `matches` devuelve `false`.

Reemplazo de subcadenas y división de cadenas

En ocasiones es conveniente reemplazar partes de una cadena, o dividir una cadena en varias piezas. Para este fin la clase `String` proporciona los métodos **`replaceAll`**, **`replaceFirst`** y **`split`**. Estos métodos se muestran en la figura 14.23.

```
1  // Fig. 14.23: SustitucionRegex.java
2  // Uso de los métodos replaceFirst, replaceAll y split.
3  import java.util.Arrays;
4
5  public class SustitucionRegex
6  {
7     public static void main(String[] args)
8     {
9        String primeraCadena = "Este enunciado termina con 5 estrellas *****";
10       String segundaCadena = "1, 2, 3, 4, 5, 6, 7, 8";
11
12       System.out.printf("Cadena 1 original: %s%n", primeraCadena);
13
14       // sustituye '*' con '^'
15       primeraCadena = primeraCadena.replaceAll("\\*", "^");
16
17       System.out.printf("^ sustituyen a *: %s%n", primeraCadena);
```

Fig. 14.23 | Métodos `replaceFirst`, `replaceAll` y `split` de `String` (parte 1 de 2).

```
18
19          // sustituye 'estrellas' con 'intercaladores'
20          primeraCadena = primeraCadena.replaceAll("estrellas", "intercaladores");
21
22          System.out.printf(
23             "\"intercaladores\" sustituye a \"estrellas\": %s%n", primeraCadena);
24
25          // sustituye las palabras con 'palabra'
26          System.out.printf("Cada palabra se sustituye por \"palabra\": %s%n%n",
27             primeraCadena.replaceAll("\\w+", "palabra"));
28
29          System.out.printf("Cadena 2 original: %s%n", segundaCadena);
30
31          // sustituye los primeros tres dígitos con 'digito'
32          for (int i = 0; i < 3; i++)
33             segundaCadena = segundaCadena.replaceFirst("\\d", "digito");
34
35          System.out.printf(
36          "Los primeros 3 digitos se sustituyeron por \"digito\" : %s%n", segundaCadena);
37
38          System.out.print("Cadena dividida en comas: ");
39          String[] resultados = segundaCadena.split(",\\s*"); // se divide en las comas
40          System.out.println(Arrays.toString(resultados));
41       }
42    } // fin de la clase SustitucionRegex
```

```
Cadena 1 original: Este enunciado termina con 5 estrellas *****
^ sustituyen a *: Este enunciado termina con 5 estrellas ^^^^^
"intercaladores" sustituye a "estrellas": Este enunciado termina con 5 intercala-
dores ^^^^^
Cada palabra se sustituye por "palabra": palabra palabra palabra palabra palabra
palabra ^^^^^

Cadena 2 original: 1, 2, 3, 4, 5, 6, 7, 8
Los primeros 3 digitos se sustituyeron por "digito" : digito, digito, digito, 4, 5,
6, 7, 8
Cadena dividida en comas: ["digito", "digito", "digito", "4", "5", "6", "7", "8"]
```

Fig. 14.23 | Métodos `replaceFirst`, `replaceAll` y `split` de `String` (parte 2 de 2).

El método `replaceAll` reemplaza el texto en un objeto `String` con nuevo texto (el segundo argumento) en cualquier parte en donde el objeto `String` original concuerde con una expresión regular (el primer argumento). En la línea 15 se reemplaza cada instancia de "*" en `primeraCadena` con "^". La expresión regular ("*") coloca dos barras diagonales inversas (\) antes del carácter *. Por lo general, * es un cuantificador que indica que una expresión regular debe concordar con *cualquier número de ocurrencias* del patrón que se coloca antes de este carácter. Sin embargo, en la línea 15 queremos encontrar todas las ocurrencias del carácter literal *; para ello debemos escapar el carácter * con el carácter \. Al escapar un carácter especial de expresión regular con una \, indicamos al motor de concordancia de expresiones regulares que busque el carácter en sí. Como la expresión está almacenada en un objeto `String` de Java y \ es un carácter especial en los objetos `String` de Java, debemos incluir un \ adicional. Por lo tanto, la cadena de Java "*" representa el patrón de expresión regular *, que concuerda con un solo carácter * en la cadena de búsqueda. En la línea 20, todas las coincidencias con la expresión regular "`estrellas`" en `primeraCadena` se reemplazan con "`intercaladores`". En la línea 27 se utiliza `replaceAll` para reemplazar todas las palabras en la cadena con "`palabra`".

El método replaceFirst (línea 33) reemplaza la primera ocurrencia de la concordancia de un patrón. En Java los objetos String son inmutables; por lo tanto, el método replaceFirst devuelve un nuevo objeto String en el que se han reemplazado los caracteres apropiados. Esta línea toma el objeto String original y lo reemplaza con el objeto String devuelto por replaceFirst. Al iterar tres veces, reemplazamos las primeras tres instancias de un dígito (\d) en segundaCadena con el texto "digito".

El método split divide un objeto String en varias subcadenas. La cadena original se divide en cualquier posición que concuerde con una expresión regular especificada. El método split devuelve un arreglo de objetos String que contiene las subcadenas que resultan de cada concordancia con la expresión regular. En la línea 39 utilizamos el método split para descomponer en tokens un objeto String de enteros separados por comas. El argumento es la expresión regular que localiza el delimitador. En este caso, utilizamos la expresión regular ",\\s*" para separar las subcadenas siempre que haya una coma. Al concordar con cualquier carácter de espacio en blanco, eliminamos los espacios adicionales de las subcadenas resultantes. Las comas y los espacios en blanco no se devuelven como parte de las subcadenas. De nuevo el objeto String de Java ",\\s*" representa la expresión regular ,\s*. En la línea 40 se utiliza el método toString de Arrays para mostrar el contenido del arreglo resultados entre corchetes y separado por comas.

Las clases Pattern y Matcher

Además de las herramientas para el uso de expresiones regulares de la clase String, Java proporciona otras clases en el paquete java.util.regex que ayudan a los desarrolladores a manipular expresiones regulares. La clase **Pattern** representa una expresión regular. La clase **Matcher** contiene tanto un patrón de expresión regular como un objeto CharSequence en el que se va a buscar ese patrón.

CharSequence (paquete java.lang) es una interfaz que permite el acceso de lectura a una secuencia de caracteres. Esta interfaz requiere que se declaren los métodos charAt, length, subSequence y toString. Tanto String como StringBuilder implementan la interfaz CharSequence, por lo que puede usarse una instancia de cualquiera de estas clases con la clase Matcher.

Error común de programación 14.2

Una expresión regular puede compararse con un objeto de cualquier clase que implemente a la interfaz CharSequence*, pero la expresión regular debe ser un objeto* String*. Si se intenta crear una expresión regular como un objeto* StringBuilder *se produce un error.*

Si se va a utilizar una expresión regular sólo una vez, puede usarse el método static **matches** de la clase Pattern. Este método toma un objeto String que especifica la expresión regular y un objeto CharSequence en el que se va a realizar la prueba de concordancia. Este método devuelve un valor de tipo boolean, el cual indica si el objeto de búsqueda (el segundo argumento) *concuerda* con la expresión regular.

Si se va a utilizar una expresión regular más de una vez (en un ciclo, por ejemplo), es más eficiente usar el método **compile** static de la clase Pattern para crear un objeto Pattern específico de esa expresión regular. Este método recibe un objeto String que representa el patrón y devuelve un nuevo objeto Pattern, el cual puede utilizarse para llamar al método **matcher**. Este método recibe un objeto CharSequence en el que se va a realizar la búsqueda, y devuelve un objeto Matcher.

La clase Matcher cuenta con el método **matches**, el cual realiza la misma tarea que el método matches de Pattern, pero no recibe argumentos ya que el patrón y el objeto de búsqueda están encapsulados en el objeto Matcher. La clase Matcher proporciona otros métodos, incluyendo **find**, **lookingAt**, **replaceFirst** y **replaceAll**.

En la figura 14.24 presentamos un ejemplo sencillo en el que se utilizan expresiones regulares. Este programa compara las fechas de cumpleaños con una expresión regular. La expresión concuerda sólo con los cumpleaños que no ocurran en abril y que pertenezcan a personas cuyos nombres empiecen con "J".

```
1  // Fig. 14.24: ConcordanciasRegex.java
2  // Clases Pattern y Matcher.
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class ConcordanciasRegex
7  {
8     public static void main(String[] args)
9     {
10       // crea la expresión regular
11       Pattern expression =
12          Pattern.compile("J.*\\d[0-35-9]-\\d\\d-\\d\\d");
13
14       String cadena1 = "Jane nacio el 05-12-75\n" +
15          "Dave nacio el 11-04-68\n" +
16          "John nacio el 04-28-73\n" +
17          "Joe nacio el 12-17-77";
18
19       // compara la expresión regular con la cadena e imprime las concordancias
20       Matcher matcher = expression.matcher(cadena1);
21
22       while (matcher.find())
23          System.out.println(matcher.group());
24    }
25 } // fin de la clase ConcordanciasRegex
```

```
Jane nacio el 05-12-75
Joe nacio el 12-17-77
```

Fig. 14.24 | Las clases `Pattern` y `Matcher`.

En las líneas 11 y 12 se crea un objeto `Pattern` mediante la invocación al método estático `compile` de la clase `Pattern`. El *carácter de punto* "`.`" en la expresión regular (línea 12) concuerda con cualquier carácter individual, excepto un carácter de nueva línea. En la línea 20 se crea el objeto `Matcher` para la expresión regular compilada y la secuencia de concordancia (`cadena1`). En las líneas 22 y 23 se utiliza un ciclo `while` para *iterar* a través del objeto `String`. En la línea 22 se utiliza el método `find` de la clase `Matcher` para tratar de hacer que concuerde una pieza del objeto de búsqueda con el patrón de búsqueda. Cada una de las llamadas a este método empieza en el punto en el que terminó la última llamada, por lo que pueden encontrarse varias concordancias. El método `lookingAt` de la clase `Matcher` funciona de manera similar, sólo que siempre comienza desde el principio del objeto de búsqueda, y siempre encontrará la *primera* concordancia, si es que hay una.

Error común de programación 14.3

El método `matches` (de las clases `String`, `Pattern` o `Matcher`) devuelve `true` sólo si todo el objeto de búsqueda concuerda con la expresión regular. Los métodos `find` y `lookingAt` (de la clase `Matcher`) devuelven `true` si una parte del objeto de búsqueda concuerda con la expresión regular.

En la línea 23 se utiliza el método **group** de la clase `Matcher`, el cual devuelve el objeto `String` del objeto de búsqueda que concuerda con el patrón de búsqueda. El objeto `String` devuelto es el que haya concordado la última vez en una llamada a `find` o `lookingAt`. La salida en la figura 14.24 muestra las dos concordancias que se encontraron en `cadena1`.

Java SE 8

Como veremos en la sección 17.7, es posible combinar el procesamiento de expresiones regulares con lambdas y flujos de Java SE 8 para implementar poderosas aplicaciones de procesamiento de archivos y objetos `String`.

14.8 Conclusión

En este capítulo aprendió más métodos de `String` para seleccionar porciones de objetos `String` y manipularlos. También aprendió sobre la clase `Character` y sobre algunos de los métodos que declara para manejar valores `char`. En este capítulo también hablamos sobre las herramientas de la clase `StringBuilder` para crear objetos `String`. En la parte final del capítulo hablamos sobre las expresiones regulares, las cuales proporcionan una poderosa herramienta para buscar y relacionar porciones de objetos `String` que coincidan con un patrón específico. En el siguiente capítulo aprenderá acerca del procesamiento de archivos, incluyendo la forma en que se almacenan y recuperan los datos persistentes.

Resumen

Sección 14.2 Fundamentos de los caracteres y las cadenas

- El valor de una literal de carácter (pág. 597) es el valor entero del carácter en el conjunto de caracteres Unicode (pág. 597). Los objetos `String` pueden incluir letras, dígitos y varios caracteres especiales como +, -, *, / y $. Una cadena en Java es un objeto de la clase `String`. Las literales de cadena (pág. 597) se conocen por lo regular como objetos `String` y se escriben entre comillas dobles en un programa.

Sección 14.3 La clase `String`

- Los objetos `String` son inmutables (pág. 599), lo que significa que una vez que se crean, los caracteres que contienen no se pueden modificar.
- El método `length` de `String` (pág. 599) devuelve el número de caracteres en un objeto `String`.
- El método `charAt` de `String` (pág. 599) devuelve el carácter en una posición específica.
- El método `regionMatches` de `String` (pág. 601) compara la igualdad entre porciones de dos cadenas.
- El método `equals` de `String` compara la igualdad entre dos objetos. Este método devuelve `true` si el contenido de los objetos `String` es igual, y `false` en caso contrario. El método `equals` utiliza una comparación lexicográfica (pág. 602) para los objetos `String`.
- Cuando se comparan valores de tipo primitivo con ==, el resultado es `true` si ambos valores son idénticos. Cuando las referencias se comparan con ==, el resultado es `true` si ambas referencias son al mismo objeto.
- Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String`.
- El método `equalsIgnoreCase` de `String` realiza una comparación de cadenas insensible al uso de mayúsculas y minúsculas.
- El método `compareTo` de `String` usa una comparación lexicográfica y devuelve `0` si los objetos `String` que está comparando son iguales, un número negativo si la cadena con la que se invoca a `compareTo` es menor que el argumento `String`, y un número positivo si la cadena con la que se invoca a `compareTo` es mayor que el argumento `String`.
- Los métodos `startsWith` y `endsWith` de `String` (pág. 604) determinan si una cadena empieza o termina con los caracteres especificados, respectivamente.
- El método `indexOf` de `String` (pág. 605) localiza la primera ocurrencia de un carácter o de una subcadena dentro de una cadena. El método `lastIndexOf` de `String` (pág. 605) localiza la última ocurrencia de un carácter o de una subcadena dentro de una cadena.
- El método `substring` de `String` copia y devuelve parte de un objeto cadena existente.
- El método `concat` de `String` (pág. 608) concatena dos objetos cadena y devuelve un nuevo objeto cadena.

- El método `replace` de `String` devuelve un nuevo objeto cadena que reemplaza cada ocurrencia en un objeto `String` de su primer argumento carácter, con su segundo argumento carácter.
- El método `toUpperCase` de `String` (pág. 609) devuelve una nueva cadena con letras mayúsculas en las posiciones en donde la cadena original tenía letras minúsculas. El método `toLowerCase` de `String` (pág. 610) devuelve una nueva cadena con letras minúsculas en las posiciones en donde la cadena original tenía letras mayúsculas.
- El método `trim` de `String` (pág. 610) devuelve un nuevo objeto cadena en el que todos los caracteres de espacio en blanco (espacios, nuevas líneas y tabuladores) se eliminan de la parte inicial y la parte final de una cadena.
- El método `toCharArray` de `String` (pág. 610) devuelve un arreglo `char` que contiene una copia de los caracteres de una cadena.
- El método `static valueOf` de `String` devuelve su argumento convertido en una cadena.

Sección 14.4 La clase `StringBuilder`

- La clase `StringBuilder` proporciona constructores que permiten inicializar objetos `StringBuilder` sin caracteres y con una capacidad inicial de 16 caracteres, o sin caracteres y con una capacidad inicial especificada en el argumento entero, o con una copia de los caracteres del argumento `String` y una capacidad inicial equivalente al número de caracteres en el argumento `String`, más 16.
- El método `length` de `StringBuilder` (pág. 612) devuelve el número de caracteres actualmente almacenados en un objeto `StringBuilder`. El método `capacity` de `StringBuilder` (pág. 612) devuelve el número de caracteres que se pueden almacenar en un objeto `StringBuilder` sin necesidad de asignar más memoria.
- El método `ensureCapacity` de `StringBuilder` (pág. 613) asegura que un objeto `StringBuilder` tenga por lo menos la capacidad especificada. El método `setLength` de `StringBuilder` incrementa o decrementa la longitud de un objeto `StringBuilder`.
- El método `charAt` de `StringBuilder` (pág. 614) devuelve el carácter que se encuentra en el índice especificado. El método `setCharAt` (pág. 614) establece el carácter en la posición especificada. El método `getChars` de `StringBuilder` (pág. 614) copia los caracteres que están en el objeto `StringBuilder` y los coloca en el arreglo de caracteres que se pasa como argumento.
- Los métodos `append` (pág. 615) sobrecargados de la clase `StringBuilder` agregan valores de tipo primitivo, arreglos de caracteres, `String`, `Object` o `CharSequence` (pág. 615) al final de un objeto `StringBuilder`.
- Los métodos `insert` sobrecargados de la clase `StringBuilder` (pág. 617) insertan valores de tipo primitivo, arreglos de caracteres, `String`, `Object` o `CharSequence` en cualquier posición en un objeto `StringBuilder`.

Sección 14.5 La clase `Character`

- El método `isDefined` de `Character` (pág. 620) determina si un carácter está definido en el conjunto de caracteres Unicode.
- El método `isDigit` de `Character` (pág. 620) determina si un carácter es un dígito definido en Unicode.
- El método `isJavaIdentifierStart` de `Character` (pág. 620) determina si un carácter se puede utilizar como el primer carácter de un identificador en Java. El método `isJavaIdentifierPart` de `Character` (pág. 620) determina si se puede utilizar un carácter en un identificador.
- El método `isLetter` de `Character` (pág. 620) determina si un carácter es una letra. El método `isLetterOrDigit` de `Character` (pág. 620) determina si un carácter es una letra o un dígito.
- El método `isLowerCase` de `Character` (pág. 620) determina si un carácter es una letra minúscula. El método `isUpperCase` de `Character` (pág. 620) determina si un carácter es una letra mayúscula.
- El método `toUpperCase` de `Character` (pág. 620) convierte un carácter en su equivalente en mayúscula. El método `toLowerCase` de `Character` (pág. 621) convierte un carácter en su equivalente en minúscula.
- El método `digit` de `Character` (pág. 621) convierte su argumento carácter en un entero en el sistema numérico especificado por su argumento entero `raiz` (pág. 621). El método `forDigit` de `Character` (pág. 621) convierte su argumento entero `digito` en un carácter en el sistema numérico especificado por su argumento entero `raiz`.

- El método `charValue` de `Character` (pág. 622) devuelve el valor `char` almacenado en un objeto `Character`. El método `toString` de `Character` devuelve una representación `String` de un objeto `Character`.

Sección 14.6 División de objetos `String` en tokens

- El método `split` de `String` (pág. 623) descompone un objeto `String` en tokens con base en el delimitador (pág. 623) especificado como argumento, y devuelve un arreglo de objetos `String` que contiene los tokens (pág. 623).

Sección 14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

- Las expresiones regulares (pág. 624) son secuencias de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar la entrada y asegurar que lo datos se encuentren en un formato específico.
- El método `matches` de `String` (pág. 624) recibe una cadena que especifica una expresión regular y relaciona el contenido del objeto `String` en el que se llama con la expresión regular. El método devuelve un valor de tipo `boolean`, el cual indica si hubo concordancia o no.
- Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Cada clase de carácter concuerda con un solo carácter en la cadena que estamos tratando de igualar con la expresión regular.
- Un carácter de palabra (`\w`; pág. 624) es cualquier letra (mayúscula o minúscula), dígito o el carácter de guion bajo.
- Un carácter de espacio en blanco (`\s`) es un espacio, un tabulador, un retorno de carro, un carácter de nueva línea o un avance de página.
- Un dígito (`\d`) es cualquier carácter numérico.
- Para relacionar un conjunto de caracteres que no tienen una clase de carácter predefinida (pág. 624), use corchetes (`[]`). Para representar los rangos, coloque un guion corto (`-`) entre dos caracteres. Si el primer carácter en los corchetes es "`^`", la expresión acepta a cualquier carácter distinto de los que se indican.
- Cuando aparece el operador "`*`" en una expresión regular, el programa trata de relacionar cero o más ocurrencias de la subexpresión que está justo antes del "`*`".
- El operador "`+`" trata de relacionar una o más ocurrencias de la subexpresión que está antes de éste.
- El carácter "`|`" permite una concordancia de la expresión a su izquierda o a su derecha.
- Los paréntesis `()` se utilizan para agrupar partes de la expresión regular.
- El asterisco (`*`) y el signo positivo (`+`) se conocen formalmente como cuantificadores (pág. 628) .
- Un cuantificador afecta sólo a la subexpresión que va justo antes de él.
- El cuantificador signo de interrogación (`?`) concuerda con cero o con una ocurrencia de la expresión que cuantifica.
- Un conjunto de llaves que contienen un número (`{n}`) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. Si se incluye una coma después del número encerrado entre llaves, concuerda con al menos *n* ocurrencias.
- Un conjunto de llaves que contienen dos números (`{n,m}`) concuerda con entre *n* y *m* ocurrencias de la expresión que califica.
- Todos los cuantificadores son avaros (pág. 629), lo cual significa que concordarán con tantas ocurrencias como puedan, mientras que haya concordancia. Si un cuantificador va seguido de un signo de interrogación (`?`), el cuantificador se vuelve renuente (pág. 629) y concuerda con el menor número posible de ocurrencias, mientras que haya concordancia.
- El método `replaceAll` de `String` (pág. 629) reemplaza texto en una cadena con nuevo texto (el segundo argumento), en cualquier parte en donde la cadena original concuerde con una expresión regular (el primer argumento).
- Al escapar un carácter de expresión regular especial con una `\` indicamos al motor de concordancia de expresiones regulares que encuentre el carácter real, en vez de lo que éste representa en una expresión regular.
- El método `replaceFirst` de `String` (pág. 629) reemplaza la primera ocurrencia de la concordancia de un patrón y devuelve una nueva cadena en la que se han reemplazado los caracteres apropiados.
- El método `split` de `String` (pág. 629) divide una cadena en varias subcadenas en cualquier ubicación que concuerde con una expresión regular especificada, y devuelve un arreglo de las subcadenas.

- La clase `Pattern` (pág. 631) representa a una expresión regular.
- La clase `Matcher` (pág. 631) contiene tanto un patrón de expresión regular como un objeto `CharSequence`, en el cual puede buscar el patrón.
- `CharSequence` es una interfaz (pág. 631) que permite el acceso de lectura a una secuencia de caracteres. Tanto `String` como `StringBuilder` implementan a esta interfaz, por lo que se pueden utilizar con la clase `Matcher`.
- Si una expresión regular se va a utilizar sólo una vez, el método estático `matches` de `Pattern` (pág. 631) recibe una cadena que especifica la expresión regular y un objeto `CharSequence` en el que se va a realizar la concordancia. Este método devuelve un valor de tipo `boolean` que indica si el objeto de búsqueda concuerda o no con la expresión regular.
- Si una expresión regular se va a utilizar más de una vez, es más eficiente usar el método estático `compile` de `Pattern` (pág. 631) para crear un objeto `Pattern` específico para esa expresión regular. Este método recibe una cadena que representa el patrón y devuelve un nuevo objeto `Pattern`.
- El método `matcher` de `Pattern` (pág. 631) recibe un objeto `CharSequence` para realizar la búsqueda y devuelve un objeto `Matcher`. El método `matches` de `Matcher` (pág. 631) realiza la misma tarea que el método `matches` de `Pattern`, pero no recibe argumentos.
- El método `find` de `Matcher` (pág. 631) trata de relacionar una pieza del objeto de la búsqueda con el patrón de búsqueda. Cada llamada a este método empieza en el punto en el que terminó la última llamada, por lo que se pueden encontrar varias concordancias.
- El método `lookingAt` de `Matcher` (pág. 631) realiza lo mismo que `find` excepto que siempre empieza desde el inicio del objeto de búsqueda, y siempre encuentra la primera concordancia, si hay una.
- El método `group` de `Matcher` (pág. 632) devuelve la cadena del objeto de búsqueda que concuerda con el patrón de búsqueda. La cadena que se devuelve es la última que concordó mediante una llamada a `find` o a `lookingAt`.

Ejercicios de autoevaluación

14.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Cuando los objetos `String` se comparan utilizando ==, el resultado es `true` si los objetos `String` contienen los mismos valores.
b) Un objeto `String` puede modificarse una vez creado.

14.2 Para cada uno de los siguientes enunciados, escriba una instrucción que realice la tarea indicada:
a) Comparar la cadena en `s1` con la cadena en `s2` para ver si su contenido es igual.
b) Anexar la cadena `s2` a la cadena `s1`, utilizando `+=`.
c) Determinar la longitud de la cadena en `s1`.

Respuestas a los ejercicios de autoevaluación

14.1 a) Falso. Los objetos `String` se comparan con el operador == para determinar si son el mismo objeto en la memoria.
b) Falso. Los objetos `String` son inmutables y no pueden modificarse una vez creados. Los objetos `StringBuilder` sí pueden modificarse una vez creados.

14.2 a)
```
s1.equals(s2)
```
b)
```
s1 += s2;
```
c)
```
s1.length()
```

Ejercicios

14.3 ***(Comparación de objetos `String`)*** Escriba una aplicación que utilice el método `compareTo` de la clase `String` para comparar dos cadenas introducidas por el usuario. Muestre si la primera cadena es menor, igual o mayor que la segunda.

14.4 ***(Comparar porciones de objetos `String`)*** Escriba una aplicación que utilice el método `regionMatches` de la clase `String` para comparar dos cadenas introducidas por el usuario. La aplicación deberá recibir como entrada el número de caracteres a comparar y el índice inicial de la comparación. La aplicación deberá indicar si las cadenas son iguales. Ignore si los caracteres están en mayúsculas o en minúsculas al momento de realizar la comparación.

14.5 ***(Enunciados aleatorios)*** Escriba una aplicación que utilice la generación de números aleatorios para crear enunciados. Use cuatro arreglos de cadenas llamados `articulo`, `sustantivo`, `verbo` y `preposicion`. Cree una oración seleccionando una palabra al azar de cada uno de los arreglos, en el siguiente orden: `articulo`, `sustantivo`, `verbo`, `preposicion`, `articulo` y `sustantivo`. A medida que se elija cada palabra, concaténela con las palabras anteriores en el enunciado. Las palabras deberán separarse mediante espacios. Cuando se muestre el enunciado final, deberá empezar con una letra mayúscula y terminar con un punto. El programa deberá generar y mostrar 20 enunciados en pantalla.

El arreglo de artículos debe contener los artículos "`el`", "`un`", "`algún`" y "`ningún`"; el arreglo de sustantivos deberá contener los sustantivos "`joven`", "`chico`", "`perro`", "`gato`" y "`auto`"; el arreglo de verbos deberá contener los verbos "`manejo`", "`salto`", "`corrio`", "`camino`" y "`vino`"; el arreglo de preposiciones deberá contener las preposiciones "`a`", "`desde`", "`encima de`", "`debajo de`" y "`sobre`".

14.6 ***(Proyecto: quintillas)*** Una quintilla es un verso humorístico de cinco líneas en el cual la primera y segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 14.5, escriba una aplicación en Java que produzca quintillas al azar. Mejorar el programa para producir buenas quintillas es un gran desafío, ¡pero el resultado valdrá la pena!

14.7 ***(Latín cerdo)*** Escriba una aplicación que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo:

Para formar una frase en latín cerdo a partir de una frase en español, divida la frase en palabras con el método `split` de `String`. Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras "ae". De esta forma, la palabra "salta" se convierte a "altasae", la palabra "el" se convierte en "leae" y la palabra "computadora" se convierte en "omputadoracae". Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. El método `imprimirPalabraEnLatin` deberá mostrar cada palabra. Cada token devuelto se pasará al método `imprimirPalabraEnLatin` para imprimir la palabra en latín cerdo. Permita al usuario introducir el enunciado. Use un área de texto para ir mostrando cada uno de los enunciados convertidos.

14.8 ***(Descomponer números telefónicos en tokens)*** Escriba una aplicación que reciba como entrada un número telefónico como una cadena de la forma `(555)555-5555`. La aplicación deberá utilizar el método `split` de `String` para extraer el código de área como un token, los primeros tres dígitos del número telefónico como otro token y los últimos cuatro dígitos del número telefónico como otro token. Los siete dígitos del número telefónico deberán concatenarse en una cadena. Deberán imprimirse tanto el código de área como el número telefónico. Recuerde que tendrá que modificar los caracteres delimitadores al dividir la cadena en tokens.

14.9 ***(Mostrar un enunciado con sus palabras invertidas)*** Escriba una aplicación que reciba como entrada una línea de texto, que divida la línea en tokens mediante el método `split` de `String` y que muestre los tokens en orden inverso. Use caracteres de espacio como delimitadores.

14.10 ***(Mostrar objetos `String` en mayúsculas y minúsculas)*** Escriba una aplicación que reciba como entrada una línea de texto y que la imprima dos veces; una vez en letras mayúsculas y otra en letras minúsculas.

14.11 ***(Búsqueda en objetos `String`)*** Escriba una aplicación que reciba como entrada una línea de texto y un carácter de búsqueda, y que utilice el método `indexOf` de la clase `String` para determinar el número de ocurrencias de ese carácter en el texto.

14.12 ***(Búsqueda en objetos `String`)*** Escriba una aplicación con base en el programa del ejercicio 14.11, que reciba como entrada una línea de texto y utilice el método `indexOf` de la clase `String` para determinar el número total de ocurrencias de cada letra del alfabeto en ese texto. Las letras mayúsculas y minúsculas deben contarse como una sola. Almacene los totales para cada letra en un arreglo, e imprima los valores en formato tabular después de que se hayan determinado los totales.

14.13 ***(Dividir objetos `String` en tokens y compararlos)*** Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con la letra "`b`".

14.14 ***(Dividir objetos `String` en tokens y compararlos)*** Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con las letras "`ED`".

14.15 ***(Convertir valores `int` a caracteres)*** Escriba una aplicación que reciba como entrada un código entero para un carácter y que muestre el carácter correspondiente. Modifique esta aplicación de manera que genere todos los posibles códigos de tres dígitos en el rango de 000 a 255, y que intente imprimir los caracteres correspondientes.

14.16 ***(Defina sus propios métodos de `String`)*** Escriba sus propias versiones de los métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String`.

14.17 ***(Crear objetos `String` de tres letras a partir de una palabra de cinco letras)*** Escriba una aplicación que lea una palabra de cinco letras proveniente del usuario, y que produzca todas las posibles cadenas de tres letras que puedan derivarse de las letras de la palabra con cinco letras. Por ejemplo, las palabras de tres letras producidas a partir de la palabra "trigo" son "rio", "tio" y "oir".

Sección especial: ejercicios de manipulación avanzada de cadenas

Los siguientes ejercicios son clave para el libro y están diseñados para evaluar la comprensión del lector sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de ejercicios intermedios y avanzados de manipulación de cadenas. El lector encontrará estos ejercicios desafiantes pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una hora o dos para escribir e implementar la aplicación. Otros son útiles como tareas de laboratorio que pudieran requerir dos o tres semanas de estudio e implementación. Algunos son proyectos desafiantes de fin de curso.

14.18 ***(Análisis de textos)*** La disponibilidad de computadoras con capacidades de manipulación de cadenas ha dado como resultado algunos métodos interesantes para analizar los escritos de grandes autores. Se ha dado mucha importancia para saber si realmente vivió William Shakespeare. Algunos estudiosos creen que hay una gran evidencia que indica que en realidad fue Cristopher Marlowe quien escribió las obras maestras que se atribuyen a Shakespeare. Los investigadores han utilizado computadoras para buscar similitudes en los escritos de estos dos autores. En este ejercicio se examinan tres métodos para analizar textos mediante una computadora.

a) Escriba una aplicación que lea una línea de texto desde el teclado e imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

```
Ser o no ser: ese es el dilema:
```

contiene una "a", ninguna "b", ninguna "c", etcétera.

b) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras, etcétera, que aparezcan en el texto. Por ejemplo, en la figura 14.25 se muestra la cuenta para la frase:

```
¿Qué es más noble para el espíritu?
```

c) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de ocurrencias de cada palabra distinta en el texto. La aplicación debe incluir las palabras en la tabla, en el mismo orden en el cual aparecen en el texto. Por ejemplo, las líneas:

```
Ser o no ser: ese es el dilema:

¿Qué es más noble para el espíritu?
```

contiene la palabra "ser" dos veces, La palabra "o" una vez, la palabra "ese" una vez, etcétera.

Longitud de palabra	Ocurrencias
1	0
2	2
3	1
4	2
5	0
6	2
7	1

Fig. 14.25 | La cuenta de longitudes de palabras para la cadena "`¿Qué es más noble para el espíritu?`".

14.19 ***(Impresión de fechas en varios formatos)*** Las fechas se imprimen en varios formatos comunes. Dos de los formatos más utilizados son:

```
04/25/1955 y Abril 25, 1955
```

Escriba una aplicación que lea una fecha en el primer formato e imprima dicha fecha en el segundo formato.

14.20 ***(Protección de cheques)*** Las computadoras se utilizan a menudo en los sistemas de escritura de cheques, tales como aplicaciones para nóminas y para cuentas por pagar. Existen muchas historias extrañas acerca de cheques de nómina que se imprimen (por error) con montos que se exceden de $1 millón. Los sistemas de emisión de cheques computarizados imprimen cantidades incorrectas debido al error humano o a una falla de la máquina. Los diseñadores de sistemas construyen controles en sus sistemas para evitar la emisión de dichos cheques erróneos.

Otro problema grave es la alteración intencional del monto de un cheque por alguien que planee cobrar un cheque de manera fraudulenta. Para evitar la alteración de un monto, la mayoría de los sistemas computarizados que emiten cheques emplean una técnica llamada protección de cheques. Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir un monto. Suponga que un cheque contiene ocho espacios en blanco en los cuales la computadora puede imprimir el monto de un cheque de nómina semanal. Si el monto es grande, entonces se llenarán los ocho espacios. Por ejemplo:

```
1,230.60 (monto del cheque)
-----------
12345678 (números de posición)
```

Por otra parte, si el monto es menor de $1,000, entonces varios espacios quedarían vacíos. Por ejemplo:

```
   99.87
--------
12345678
```

contiene tres espacios en blanco. Si se imprime un cheque con espacios en blanco, es más fácil para alguien alterar el monto del cheque. Para evitar que se altere el cheque, muchos sistemas de escritura de cheques insertan asteriscos al principio para proteger la cantidad, como se muestra a continuación:

```
***99.87
--------
12345678
```

Escriba una aplicación que reciba como entrada un monto a imprimir sobre un cheque y que lo escriba mediante el formato de protección de cheques, con asteriscos al principio si es necesario. Suponga que existen nueve espacios disponibles para imprimir el monto.

14.21 ***(Escritura en letras del código de un cheque)*** Para continuar con la discusión del ejercicio 14.20, reiteramos la importancia de diseñar sistemas de escritura de cheques para evitar la alteración de los montos de los cheques. Un método común de seguridad requiere que el monto del cheque se escriba tanto en números como en letras. Aun cuando alguien pueda alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en letras. Escriba una aplicación que reciba como entrada un monto numérico para el cheque que sea menor a $1,000, y que escriba el equivalente del monto en letras. Por ejemplo, el monto 112.43 debe escribirse como

```
CIENTO DOCE CON 43/100
```

14.22 ***(Clave Morse)*** Quizá el más famoso de todos los esquemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones a cada letra del alfabeto, cada dígito y algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas orientados a sonidos, el punto representa un sonido corto y el guion representa un sonido largo. Otras representaciones de puntos y guiones se utilizan en los sistemas orientados a luces y sistemas de señalización con banderas. La separación entre palabras se indica mediante un espacio o simplemente con la ausencia de un punto o un guion. En un sistema orientado a sonidos, un espacio se indica por un tiempo breve durante el cual no se transmite sonido alguno. La versión internacional del código Morse aparece en la figura 14.26.

Escriba una aplicación que lea una frase en español y que codifique la frase en clave Morse. Además, escriba una aplicación que lea una frase en código Morse y que la convierta en su equivalente en español. Use un espacio en blanco entre cada letra en clave Morse, y tres espacios en blanco entre cada palabra en clave Morse.

Carácter	Código	Carácter	Código	Carácter	Código
A	.-	N	-.	*Dígitos*	
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-	0	-----
L	.-..	Y	-.--		
M	--	Z	--..		

Fig. 14.26 | Las letras del alfabeto expresadas en código Morse internacional.

14.23 ***(Conversiones al sistema métrico)*** Escriba una aplicación que ayude al usuario a realizar conversiones métricas. Su aplicación debe permitir al usuario especificar los nombres de las unidades como cadenas (es decir, centímetros, litros, gramos, etcétera, para el sistema métrico, y pulgadas, cuartos, libras, etcétera, para el sistema inglés) y debe responder a preguntas simples tales como:

```
"¿Cuántas pulgadas hay en 2 metros?"
"¿Cuántos litros hay en 10 cuartos?"
```

Su programa debe reconocer conversiones inválidas. Por ejemplo, la pregunta:

```
"¿Cuántos pies hay en 5 kilogramos?"
```

no es correcta, debido a que los "`pies`" son unidades de longitud, mientras que los "`kilogramos`" son unidades de masa.

Sección especial: proyectos desafiantes de manipulación de cadenas

14.24 ***(Proyecto: un corrector ortográfico)*** Muchos paquetes populares de software de procesamiento de palabras cuentan con correctores ortográficos integrados. En este proyecto usted debe desarrollar su propia herramienta de corrección ortográfica. Le haremos unas sugerencias para ayudarlo a empezar. Sería conveniente que después le agregara más características. Use un diccionario computarizado (si tiene acceso a uno) como fuente de palabras.

¿Por qué escribimos tantas palabras en forma incorrecta? En algunos casos es porque simplemente no conocemos la manera correcta de escribirlas, por lo que tratamos de adivinar lo mejor que podemos. En otros casos, es porque transponemos dos letras (por ejemplo, "perdeterminado" en lugar de "predeterminado"). Algunas veces escribimos una letra doble por accidente (por ejemplo, "úttil" en vez de "útil"). Otras veces escribimos una tecla que está cerca de la que pretendíamos escribir (por ejemplo, "cunpleaños" en vez de "cumpleaños"), etcétera.

Diseñe e implemente una aplicación de corrección ortográfica en Java. Su aplicación debe mantener un arreglo de cadenas llamado `listaDePalabras`. Permita al usuario introducir estas cadenas. [*Nota:* en el capítulo 15 presentaremos el procesamiento de archivos. Con esta capacidad, puede obtener las palabras para el corrector ortográfico de un diccionario computarizado almacenado en un archivo].

Su aplicación debe pedir al usuario que introduzca una palabra. La aplicación debe entonces buscar esa palabra en el arreglo `listaDePalabras`. Si la palabra se encuentra en el arreglo, su aplicación deberá imprimir "`La palabra está escrita correctamente`". Si la palabra no se encuentra en el arreglo, su aplicación debe imprimir "`La palabra no está escrita correctamente`". Después su aplicación debe tratar de localizar otras palabras en la `listaDePalabras` que puedan ser la palabra que el usuario trataba de escribir. Por ejemplo, puede probar con todas las transposiciones simples posibles de letras adyacentes para descubrir que la palabra "predeterminado" concuerda directamente con una palabra en `listaDePalabras`. Desde luego que esto implica que su programa comprobará todas las otras transposiciones posibles, como "rpedeterminado", "perdeterminado", "predetreminado", "predeterminado" y "predetermniado". Cuando encuentre una nueva palabra que concuerde con una en la `listaDePalabras`, imprima esa palabra en un mensaje como

```
"¿Quiso decir "predeterminado"?"
```

Implemente otras pruebas, como reemplazar cada letra doble con una sola letra y cualquier otra prueba que pueda desarrollar para aumentar el valor de su corrector ortográfico.

14.25 ***(Proyecto: un generador de crucigramas)*** La mayoría de las personas han resuelto crucigramas pero pocos han intentado generar uno. Aquí lo sugerimos como un proyecto de manipulación de cadenas que requiere una cantidad considerable de sofisticación y esfuerzo.

Hay muchas cuestiones que el programador tiene que resolver para hacer que funcione incluso hasta la aplicación generadora de crucigramas más simple. Por ejemplo, ¿cómo representaría la cuadrícula de un crucigrama dentro de la computadora? ¿Debería utilizar una serie de cadenas o arreglos bidimensionales?

El programador necesita una fuente de palabras (es decir, un diccionario computarizado) a la que la aplicación pueda hacer referencia de manera directa. ¿De qué manera deben almacenarse estas palabras para facilitar las manipulaciones complejas que requiere la aplicación?

Si usted es realmente ambicioso, querrá generar la porción de "claves" del crucigrama, en la que se imprimen pistas breves para cada palabra "horizontal" y cada palabra "vertical". La sola impresión de la versión del crucigrama en blanco no es una tarea fácil.

Marcando la diferencia

14.26 ***(Cocinar con ingredientes más saludables)*** La obesidad en Estados Unidos está aumentando a un ritmo alarmante. De un vistazo al mapa de los Centros para el control y la prevención de enfermedades (CDC) en `http://www.cdc.gov/obesity/data/databases.html`, que muestra las tendencias de obesidad en Estados Unidos durante los últimos 20 años. A medida que aumenta la obesidad, también se incrementan las ocurrencias de los problemas rela-

cionados (enfermedades cardiacas, presión alta, colesterol alto, diabetes tipo 2). Escriba un programa que ayude a los usuarios a elegir ingredientes más saludables al cocinar, y que ayude a los que padecen alergias a ciertos alimentos (como nueces, gluten, etc.) a encontrar sustitutos. El programa debe leer una receta de un objeto `JTextArea` y sugerir sustitutos más saludables para algunos de los ingredientes. Por cuestión de simpleza, su programa debe asumir que la receta no tiene abreviaciones para las medidas como cucharaditas, tazas y cucharadas, y que usa dígitos numéricos para las cantidades (por ejemplo, 1 huevo, 2 tazas) en vez de deletrearlos (un huevo, dos tazas). Algunos sustitutos comunes se muestran en la figura 14.27. Su programa debe mostrar una advertencia tal como, "Consulte siempre a su médico antes de realizar cambios considerables en su dieta".

Su programa debe tener en cuenta que los sustitutos no son siempre en la misma cantidad. Por ejemplo, si la receta de un pastel dice que se usan tres huevos, una sustitución razonable podría ser seis claras de huevo. Es posible obtener datos de conversión para las medidas y sustitutos en sitios Web tales como:

```
chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm
```

Su programa debe considerar los problemas de salud del usuario como el colesterol alto, la presión alta, la pérdida de peso, la alergia al gluten, etcétera. Para el colesterol alto, el programa debe sugerir sustitutos para huevos y productos lácteos; si el usuario desea perder peso, deberían sugerirse sustitutos de bajas calorías para los ingredientes como el azúcar.

Ingrediente	Sustituto
1 taza de crema agria	1 taza de yogur
1 taza de leche	1/2 taza de leche evaporada y 1/2 taza de agua
1 cucharadita de jugo de limón	1/2 cucharadita de vinagre
1 taza de azúcar	1/2 taza de miel, 1 taza de melaza o 1/4 taza de néctar de agave
1 taza de mantequilla	1 taza de margarina o yogur
1 taza de harina	1 taza de harina de centeno o de arroz
1 taza de mayonesa	1 taza de queso *cottage* o 1/8 taza de mayonesa y 7/8 taza de yogur
1 huevo	2 cucharadas de almidón, harina de arrurruz o almidón de papa, o 2 claras de huevo, o 1/2 plátano grande (machacado)
1 taza de leche	1 taza de leche de soya
1/4 taza de aceite	1/4 taza de puré de manzana
pan blanco	pan de grano entero

Fig. 14.27 | Sustitutos comunes de alimentos.

14.27 ***(Explorador de spam)*** El spam (o correo electrónico basura) cuesta a las organizaciones estadounidenses miles de millones de dólares al año en software para prevención de spam, equipo, recursos de red, ancho de banda y pérdida de productividad. Investigue en línea algunos de los mensajes y palabras de correo electrónico de spam más comunes, y revise su propia carpeta de correo electrónico basura. Cree una lista de 30 palabras y frases que se encuentren comúnmente en los mensajes de spam. Escriba una aplicación en donde el usuario introduzca un mensaje de texto en un objeto `JTextArea`. Después explore el mensaje en busca de cada una de las 30 palabras clave o frases. Para cada ocurrencia de una de éstas dentro del mensaje, agregue un punto a la "puntuación del spam"

del mensaje. Después califique la probabilidad de que el mensaje sea spam, con base en el número de puntos que haya recibido.

14.28 ***(Lenguaje SMS)*** El Servicio de mensaje cortos (SMS) es un servicio de comunicaciones que permite enviar mensajes de texto de 160 o menos caracteres entre teléfonos móviles. Con la proliferación del uso de teléfonos móviles en todo el mundo, SMS se utiliza en muchas naciones en desarrollo para fines políticos (por ejemplo, manifestar opiniones y oposición), informar noticias sobre desastres naturales, etcétera. Por ejemplo, visite `comunica.org/radio2.0/archives/87`. Como la longitud de los mensajes SMS es limitada, a menudo se utiliza el lenguaje SMS que consta de abreviaciones de palabras y frases comunes en mensajes de texto móviles, correos electrónicos, mensajes instantáneos, etc. Por ejemplo, "ntp" es "no te preocupes" en lenguaje SMS. Investigue el lenguaje SMS en línea. Escriba una aplicación de GUI en donde el usuario pueda introducir un mensaje mediante lenguaje SMS, y que después haga clic en un botón para traducirlo en español (o en su propio lenguaje). Proporcione además un mecanismo para traducir texto escrito en español a lenguaje SMS. Un posible problema es que una abreviación en SMS podría expandirse en una variedad de frases. Por ejemplo, "ntp" (como se indicó antes) podría también representar "nada te parece", "no tengo prisa", o cualquier otra cosa.

El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#,Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas…

VISITA

www.detodoprogramacion.org

www.detodopython.com

Detodo Python.com