Programación Avanzada
UD4

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

ÍNDICE GENERAL

CAPÍTULO 4

ARRAYS Y STRINGS

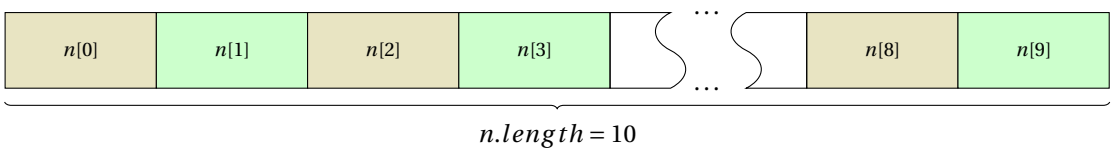
En este capítulo se explica el uso de matrices o arrays, y el de cadenas de caracteres, que en Java son objetos `String`.

4.1. Matrices (arrays)

Un array es una zona de memoria que almacena un número fijo de valores de un mismo tipo (estructura homogénea), el cual puede ser tanto un tipo primitivo como una clase. Todos los valores se almacenan de forma consecutiva en la memoria, reservándose la memoria necesaria para almacenar el número de elementos especificado.

Declaración y creación de arrays

A diferencia del lenguaje C, el tamaño del array no se especifica cuando se declara, sino cuando se crea, ya que los arrays en Java son objetos, y es necesario crearlos mediante el operador `new`, momento en el que se indica el número de elementos que tendrá. Sin embargo, la forma de utilizar los arrays en Java es similar al lenguaje C; podremos acceder a cada elemento directamente con el nombre de la variable matriz seguido de un índice encerrado entre corchetes `[...]` (o más índices en el caso arrays multi-dimensionales). Asimismo, el tamaño ya será fijo, no podrá modificarse. La siguiente imagen muestra un array de 10 elementos: `n[0]..n[9]`.



En los siguientes ejemplos se muestra como declarar y crear arrays, en este caso de tipo base `int`:

```
// Declaración de un array
int m[];
// Creación
m = new int[3];
```

```
// Declaración de dos arrays
int[] m, n;
// Creación
m = new int[3];
n = new int[10];
```

Podemos observar que al declarar un array es indistinto poner los corchetes detrás del nombre del array o detrás del tipo; este último caso es más recomendado, además de que es más conveniente cuando se van a declarar varios arrays del mismo tipo, ya que evita tener que escribir los corchetes en cada uno.

Al igual que podemos inicializar una variable local al mismo tiempo de su declaración, también podemos declarar y crear un array al mismo tiempo, incluso también es posible inicializarlo a unos valores concretos.

```
// Declaración y creación
int m[] = new int[3];
// Inicializacion de valores
m[0] = 1;  m[1] = 2;  m[2] = 3;
```

```
// Declaración, creación e
    inicializacion
int m[] = { 1, 2, 3 };
```

El siguiente ejemplo crea un array con unos valores concretos, y posteriormente recorre todo el array para obtener su suma. Se utiliza la variable `length` que tiene cada array, la cual nos permite conocer su tamaño, es decir, el número de elementos con el que se creó.

```
1  // Creacion de un array e inicializacion con unos valores
2  class ArraysDemo1 {
3     public static void main (String args[]) {
4        int suma = 0;
5        int[] m;
6
7        m = new int[3];
8        m[0] = 1;
9        m[1] = 2;
10       m[2] = 3;
11       for (int i=0; i < m.length; i++) {
12          suma = suma + m[i];
13          System.out.println ("Elemento m[" + i + "] = " + m[i]);
14       }
15       System.out.println ("Suma = " + suma);
16    }
17 }
```

```
Elemento m[0] = 1
Elemento m[1] = 2
Elemento m[2] = 3
Suma = 6
```


Cuando se crea un array, todos los elementos se inicializan automáticamente al valor por defecto según el tipo (ver tabla 4.1). Sin embargo, en el caso de que el array sea de objetos, es decir, el tipo base sea una clase, no se crea ningún objeto, sino que cada elemento del array se inicializa al valor `null`, es decir, cada elemento será una referencia a un objeto no creado (referencia nula). Por ejemplo, si se crea un array de cadenas de caracteres, como éstas son objetos de la clase `String`, el array se inicializará a null, no a la cadena vacía ‘“’ o cualquier otra cadena; habrá que crear explícitamente cada una de ellas.

Tipo	Valor por defecto
`char`	’\0’ o ’\$u$0000’
`byte`	0
`short`	0
`int`	0
`long`	0L
`float`	0.0F
`double`	0.0D
`boolean`	`false`
Objetos (p.e. `String`)	`null`

Cuadro 4.1: Valores por defecto para cada tipo primitivo y objetos

También es posible utilizar arrays multidimensionales. Los siguientes ejemplos declaran y crean un array de 2 dimensiones de dos posibles formas, asignando los mismos valores a todos los elementos. Como se ha dicho antes, los corchetes pueden ir detrás del nombre de la variable o detrás del tipo al declararse.

```
// Array de 2 dimensiones: 2x3
int n[][] = new int[2][3];
n[0][0]=1; n[0][1]=2; n[0][2]=3;
n[1][0]=4; n[1][1]=5; n[1][2]=6;
```

```
// Array de 2 dimensiones: 2x3
int[][] n = {
  { 1, 2, 3 },
  { 4, 5, 6 } };
```

Copiar arrays

Una forma de copiar un array en otro sería de forma manual mediante un bucle. Sin embargo, existe una forma mucho más eficiente, utilizar el método estático `arraycopy()` de la clase `System` (ver tabla 4.2), el cual nos permite copiar todo un array o solo una parte. Puesto que la clase `System` pertenece al paquete `java.lang`, no es necesario importar este paquete, ya que se importa automáticamente, tal como se explicará en el capítulo correspondiente a paquetes.

Método	Descripción
`static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`	Copia un array `src` o subrango de él en otro array `dest`, comenzando en la posición `srcPos` indicada del array origen, y tantos elementos como se indique en `length`. También es posible indicar la dirección del array destino `destPost` en la que comenzar a escribir.

Cuadro 4.2: Clase `System` - método para copiar arrays

El método espera que el objeto origen y destino recibidos sean arrays, los cuales deben ser del mismo tipo base. Además, deben tener un tamaño apropiado para copiar los elementos especificados con los otros argumentos, de lo contrario obtendremos un error de ejecución. En caso de que alguno de los parámetros no sean correcto se pueden generar las siguientes **excepciones**:

- `ArrayStoreException`: si alguno de los objetos especificados como array origen o destino no son arrays, o lo son, pero su tipo base es diferente.
- `NullPointerException`: si alguno de los objetos especificados como array origen o destino no se hayan creado (son `null`).
- `IndexOutOfBoundsException`: si los indices especificados como índices origen o destino sumando la longitud especificada se exceden los límites de alguno de los arrays.

El siguiente ejemplo declara un array de `char`, que luego copia dos veces en otro utilizando el método `arraycopy()`. Finalmente se imprimen las dos cadenas.

```
class ArraysDemo2 {
    public static void main(String[] args) {
        char[] src = { 'E', 'n', ' ', 'u', 'n', ' ',
                       'l', 'u', 'g', 'a', 'r', ' ',
                       'd', 'e', ' ', 'L', 'a', ' ',
                       'M', 'a', 'n', 'c', 'h', 'a' };

        // Creamos el array destino con el doble de tamaño que el origen
        char[] dst = new char[src.length*2];
        // Copiamos todos los elementos dos veces
        System.arraycopy (src, 0, dst, 0, src.length);
        System.arraycopy (src, 0, dst, src.length, src.length);
        // Imprimimos el array fuente
        System.out.print ("Fuente : ");
        for (int i=0; i<src.length; i++) {
           System.out.print (src[i]);
        }
        System.out.println ("");
        // Imprimimos el array destino
        System.out.print ("Destino: ");
        for (int i=0; i<dst.length; i++) {
           System.out.print (dst[i]);
        }
        System.out.println ("");
    }
}
```

```
Fuente : En un lugar de La Mancha
Destino: En un lugar de La ManchaEn un lugar de La Mancha
```


En la siguiente sección se explica la clase `Arrays`, la cual también dispone de dos métodos para copiar dos arrays. También dispone de métodos para convertir a cadena, lo cual simplifica la tarea de imprimir un array.

Manipulación de arrays: clase `Arrays`

Java dispone de la clase `Arrays`, específica para manipular matrices. Puesto que esta clase se encuentra dentro del paquete `java.util`, es necesario importarlo previamente; aunque la gestión de paquetes se verá en otro capítulo, esto se haría escribiendo la siguiente línea al principio del fichero donde se necesite utilizar la clase, fuera incluso de la clase:

```
1 import java.util.Arrays;
2
3 class MiClase {
4     ...
5 }
```

La clase `Arrays` dispone de una serie de métodos estáticos para la manipulación de arrays. En la tabla 4.3 se muestra casi todos los métodos disponibles en la clase, pero con puntos suspensivos como argumentos, ya que están sobrecargadas (polimorfismo), es decir, existen diversas versiones para arrays con cualquier tipo base: `int`, `float`, `...`, y objetos (`Object`); consultar la documentación en línea de la librería de clases de Java (API Specification) para más detalles.

Método	Descripción
`static int binarySearch(...)`	Buscar un valor dentro del array utilizando el algoritmo de búsqueda binaria que tiene una gran eficiencia. Retorna la posición del valor buscado (>=0), o un valor negativo (<0) si no existe.
`static ... copyOf(...)`	Copia un array en otro con la longitud especificada, truncando o rellenando con el valor por defecto si fuese necesario.
`static ... copyOfRange(...)`	Copia un array en otro desde una posición inicial hasta una final (rango).
`static boolean equals(...)`	Compara si dos arrays son iguales, comprobando que tengan el mismo tipo base, el mismo número de elementos, y comparando el cada par de elementos de la misma posición.
`static void fill(...)`	Inicializa todos o un rango de valores del array con un valor especificado.
`static void sort(...)`	Ordena los valores del array en orden ascendente mediante un algoritmo basado en el *quicksort*.
`static String toString(...)`	Convierte a cadena el contenido del array. El resultado es un objeto String compuesto por la representación en forma de cadena de cada elemento, separados por una coma y un espacio, todo ello envuelto entre corchetes, es decir, una cadena del estilo: `[1, 2, 3, ...]`.

Cuadro 4.3: Clase `Arrays` - métodos para manipulación de arrays

Además de los métodos anteriores para copiar un array podemos utilizar el método `arraycopy()`, el cual se presentó en la sección anterior. Este se vuelve a utilizar en el siguiente ejemplo, junto con otros de la clase `Arrays`.

Como ejemplo de manipulación de arrays, el siguiente ejemplo crea un array de enteros de un número de elementos concreto, y lo inicializa con una serie de valores generados al azar entre 0..9. Para comprobar los valores generados se imprime el array, convirtiendolo a cadena con `Arrays.toString()`. Después lo copia en un segundo array, lo imprime, y muestra si son iguales (en este caso sí que lo serán). Después inicializa a 1 todos los elementos del segundo array, y vuelve a compararlos (ahora ya no serán iguales).

```
1  import java.util.Arrays;
2
3  public class ArraysManipulacion {
4
5     private static final int NUM_ELEMENTOS = 5;
6
7     public static void main (String args[]) {
8        int[] m1, m2;  // Declaramos 2 arrays
9
10       // Creamos el primer array
11       m1 = new int[NUM_ELEMENTOS];
12       // Inicializamos el primer array con valores aleatorios 0..9
13       for (int i=0; i < m1.length; i++) {
14          m1[i] = (int) (Math.random()*10.0);
15       }
16       // Imprimimos los valores del primer array
17       System.out.println ("m1 = " + Arrays.toString(m1));
18
19       // Copiamos el primer array en en el segundo
20       m2 = new int[m1.length];
21       System.arraycopy (m1, 0, m2, 0, m1.length);
22       // Imprimimos los valores del segundo array (copia)
23       System.out.println ("m2 = " + Arrays.toString(m2));
24
25       // Comparamos los dos arrays (deben ser iguales)
26       System.out.println("¿Iguales? = " + Arrays.equals(m1,m2));
27
28       // Inicializamos el segundo array con el valor 1 en todos
29       Arrays.fill (m2, 1);
30       // Imprimimos los nuevos valores del segundo array (todo 1)
31       System.out.println ("m2 = " + Arrays.toString(m2));
32
33       // Volvemos a comparar los dos arrays (ahora son diferentes)
34       System.out.println("¿Iguales? = " + Arrays.equals(m1,m2));
35    }
36 }
```


El resultado de la ejecución de este programa será:

```
m1 = [0, 7, 8, 6, 9]
m2 = [0, 7, 8, 6, 9]
¿Iguales? = true
m2 = [1, 1, 1, 1, 1]
¿Iguales? = false
```

Notar el uso que se hace del operador suma `'+'` para concatenar varias cadenas de caracteres y valores numéricos, para convertirlo todo en una cadena que se imprime con el método `println()` del flujo de salida `'out'` que representa la pantalla.

Para terminar con el apartado de *arrays*, vamos a utilizar el bucle `for-each` (introducido en la versión Java 5) para recorrer todos sus elementos de una forma más compacta que con el bucle `for` tradicional. El bucle `for-each` se presentó en un capítulo anterior, y es válido tanto para arrays como para objetos de colecciones (que se verán en un capítulo posterior).

```
1  public class ArraysDemo3 {
2
3      public static void main (String args[]) {
4          int[] n = { 1, 2, 3, 4, 5};
5
6          // Recorremos todos los elementos
7          for (int i : n) {
8              System.out.print (i + " ");
9          }
10         System.out.println ("");
11     }
12 }
```

Su ejecución dará como resultado:

```
1 2 3 4 5
```


4.2. Cadenas de caracteres

A lo largo de los capítulos anteriores ya se han utilizado cadenas de caracteres, que son una secuencia de caracteres de tipo `char` (Unicode). Si un carácter `char` se escribe entre comillas simples (p.e. `'h'`), una cadena de caracteres se escribe entre comillas dobles (p.e. `"hola"`). En Java, las cadenas de caracteres son objetos de la clase `String`, por lo que deberían crearse como cualquier otro objeto, es decir, mediante el operador `new` que invoque a algún constructor, pero como ya se comentó, al ser las cadenas de caracteres muy utilizadas se simplificó el lenguaje para permitir una forma más compacta.

Las siguientes sentencias crean la misma cadena de caracteres de forma equivalente:

```
1 // 1. Forma compacta especifica para cadenas
2 String hola1 = "Hola";
3
4 // 2. Utilizando un constructor desde otra cadena
5 String hola2 = new String ("Hola");
6
7 // 3. A partir de un array de char
8 char holaArray[] = { 'H', 'o', 'l', 'a' };
9 String hola3 = new String (holaArray);
```

En los dos últimos casos vemos que se han utilizado dos constructores diferentes; la clase `String` ofrece multitud de constructores: para crear una cadena vacía, desde un array de `byte`, desde un array de `char`, desde otro `String`, ... Los más utilizados se resumen en la tabla 4.4, aunque hay más (unos 15).

Constructor	Descripción
`String ()`	Crea un `String` con la cadena vacía `""`.
`String (bytes[] s)`	Crea un `String` con los códigos Unicode de un array de `byte`.
`String (char[] s)`	Crea un `String` con los caracteres del array.
`String (String s)`	Crea un `String` desde otro `String`, que normalmente será una cadena de caracteres constante.
`String (StringBuffer s)`	Crea un `String` desde un objeto `StringBuffer`, que es un `String` modificable, no inmutable.

Cuadro 4.4: Clase `String` - constructores

Una vez creado un objeto `String`, es posible realizar ciertas acciones, como averiguar su longitud con `length()`, obtener el carácter de una determinada posición, comparar la cadena con otra con `equals()`, etc. Los métodos más interesantes se muestran en la tabla 4.5, sin ningún orden concreto, tan sólo agrupados por similitud en su operación.

Hay algunos métodos que aparentemente realizan alguna transformación en la cadena, como convertirla a minúsculas con `toLowerCase()` o a mayúsculas con `toUpperCase()`, o sustituir un caracter por otro con `replace()`. En realidad, lo que hacen es retornar una nueva cadena realizando la transformación solicitada, pero dejando la cadena original sin cambios (inmutable). En siguientes secciones se explicará que los objetos `String` son inmutables, así como el uso de la clase `StringBuffer` para el caso de necesitar la modificación de las cadenas.

Método	Descripción
`char charAt (int index)`	Retorna el carácter `char` de la posición indicada, que deberá estar comprendida entre 0 y su longitud-1.
`String concat (String s)`	Devuelve un nuevo objeto `String` resultado de concatenar la cadena pasada como argumento al final de la cadena actual. También podemos concatenar dos objetos `String` con los operadores + y +=.
`boolean isEmpty ()`	Retorna true si la cadena es de longitud 0.
`int length ()`	Devuelve la longitud de la cadena actual, es decir, el número de caracteres Unicode.
`int compareTo (String s)`	Compara lexicográficamente dos cadenas de caracteres, retornando: 0 si son iguales, <0 si el objeto `String` que invoca al método es menor que el objeto `String` que se pasa como argumento, y >0 si es mayor, según el orden lexicográfico.
`int compareToIgnoreCase (String s)`	Idem al anterior, pero ignorando mayúsculas o minúsculas.
`boolean equals (Object o)`	Compara si dos cadenas son iguales, la actual con la que recibe como parámetro (es un `Object` que se convierte a `String`).
`boolean equalsIgnoreCase (String s)`	Idem al anterior, sin tener en cuenta las mayúsculas o minúsculas.
`boolean startsWith (String s)`	Comprueba si la cadena actual empieza con la cadena `s`.
`boolean endsWith (String s)`	Comprueba si la cadena actual termina con la cadena `s`.
`int indexOf (int c [,int desde])`	Busca la posición del carácter especificado en la cadena actual comenzando por el principio (por defecto) o desde una posición, hacia el final; retorna -1 si no la encuentra.
`int indexOf (String s [,int desde])`	Idem al anterior, pero busca la cadena especificada.
`int lastIndexOf (int c [, int desde])`	Busca la posición del carácter especificado en la cadena actual comenzando por el final (por defecto) o desde una posición, hacia el principio; retorna -1 si no la encuentra.
`int lastIndexOf (String s [, int desde])`	Idem al anterior, pero busca la cadena especificada.
`String substring (int inicio [, int fin])`	Retorna una nueva cadena con un fragmento o subcadena de la cadena actual, comprendida entre la posición `inicio` y el final, o desde la posición `inicio` hasta la posición `fin` si se especifica.
`String toLowerCase ()`	Retorna una nueva cadena transformando a minúsculas todas las letras de la cadena actual.
`String toUpperCase ()`	Retorna una nueva cadena transformando a mayúsculas todas las letras de la cadena actual.
`char[] toCharArray ()`	Convierte la cadena actual en un array de caracteres `char`.
`String trim ()`	Retorna una nueva cadena eliminando los espacios en blanco del principio y del final.
`String replace (char old, char new)`	Retorna una nueva cadena sustituyendo todas las ocurrencias de un carácter por otro en la cadena actual.
`String valueOf (...)`	Transforma el valor especificado (el cual puede ser de cualquier tipo primitivo o un objeto de cualquier clase) a cadena de caracteres.
`String intern()`	Añadir la cadena actual a la zona reservada para cadenas (caso de que no exista), y retornar la referencia a dicha cadena.

Cuadro 4.5: Clase `String` - métodos

El siguiente ejemplo hace uso de algunos de los métodos anteriores.

```
1  public class StringManipulacion {
2     public static void main (String args[]) {
3        String s, s2, sMin, sMay;
4
5        // Mostramos la cadena original;
6        s = " En un lugar de La Mancha ";
7        System.out.println ("Cadena original  : " + s +
8                            " Longitud: " + s.length());
9
10       // Eliminamos los espacios en blanco iniciales y finales
11       s2 = s.trim ();
12       System.out.println ("Cadena recortada : " + s2 +
13                           " Longitud: " + s2.length());
14
15       // La transformamos a minusculas y mayusculas
16       sMin = s2.toLowerCase ();
17       sMay = s2.toUpperCase ();
18       System.out.println ("Cadena minusculas: " + sMin +
19                           " Longitud: " + sMin.length());
20       System.out.println ("Cadena mayusculas: " + sMay +
21                           " Longitud: " + sMay.length());
22
23       // Averiguamos si la cadena en minusculas y mayusculas son iguales
24       System.out.println ("¿Iguales distinguiendo mayusculas?  " +
25                           (sMin.equals (sMay) ? "si" : "no"));
26       System.out.println ("¿Iguales sin distinguir mayusculas? " +
27                           (sMin.equalsIgnoreCase (sMay) ? "si" : "no"));
28
29       // En la cadena en minusculas buscamos la letra ele
30       int posIni = sMin.indexOf ('l');
31       int posFin = sMin.lastIndexOf ('l');
32       String subcadena = sMin.substring (posIni, posFin);
33       // Mostramos la subcadena comprendida entre esos dos indices
34       System.out.println ("Subcadena: " + subcadena +
35                           " Longitud: " + subcadena.length());
36
37       // Averiguamos si la subcadena empieza por la palabra "lugar"
38       boolean comienzapor = subcadena.startsWith ("lugar");
39       System.out.println ("¿Subcadena comienza por \"lugar\"? " +
40                           (comienzapor ? "si" : "no"));
41    }
42 }
```


Tras ejecutar el programa anterior obtendremos el siguiente resultado:

```
Cadena original  :  En un lugar de La Mancha  Longitud: 26
Cadena recortada : En un lugar de La Mancha Longitud: 24
Cadena minusculas: en un lugar de la mancha Longitud: 24
Cadena mayusculas: EN UN LUGAR DE LA MANCHA Longitud: 24
¿Iguales distinguiendo mayusculas?  no
¿Iguales sin distinguir mayusculas? si
Subcadena: lugar de  Longitud: 9
¿Subcadena comienza por "lugar"? si
```

En el siguiente ejemplo haremos transformaciones entre objetos `String` y arrays de tipo `char`.

```
1  import java.util.Arrays;
2
3  public class StringManipulacion2 {
4     public static void main (String args[]) {
5        String s1, s2;
6        char[] lugar1 = { 'e', 'n', ' ', 'u', 'n', ' ',
7                          'l', 'u', 'g', 'a', 'r', ' ',
8                          'd', 'e', ' ', 'l', 'a', ' ',
9                          'm', 'a', 'n', 'c', 'h', 'a' };
10       char[] lugar2, lugar3;
11
12       // Creamos un String desde un array de char
13       s1 = new String (lugar1);
14       System.out.println ("Array1: " + Arrays.toString(lugar1) +
15                           " Longitud: " + lugar1.length);
16       System.out.println ("Cadena original: " + s1 +
17                           " Longitud: " + s1.length());
18
19       // Sustituimos todas las 'u' y 'a' por la letra 'e'
20       s2 = s1.replace ('u', 'e');   s2 = s2.replace ('a', 'e');
21       System.out.println ("Cadena nueva   : " + s2 +
22                           " Longitud: " + s2.length());
23
24       // Transformamos el String a un array (2 formas)
25       lugar2 = s2.toCharArray ();
26       System.out.println ("Array2: " + Arrays.toString(lugar2) +
27                           " Longitud: " + lugar2.length);
28       lugar3 = new char[s2.length()];
29       for (int i=s2.length()-1, j=0; i>=0; i--, j++) {
30          lugar3[j] = s2.charAt (i);
31       }
32       System.out.println ("Array3: " + Arrays.toString(lugar3) +
33                           " Longitud: " + lugar3.length);
34    }
35 }
```


Primero creamos un `String` desde un array de `char`, después sustituimos algunas letras en la cadena. Luego transformamos la cadena final a un array mediante `toCharArray()`, y luego hacemos algo similar de forma manual, pero, invirtiendo la cadena en el momento de construir el array. Si ejecutamos este programa obtendremos lo siguiente:

```
Array1: [e, n,  , u, n,  , l, u, g, a, r,  , d, e,  , l, a,  , m, a, n, c, h, a] Longitud: 24
Cadena original: en un lugar de la mancha Longitud: 24
Cadena nueva   : en en leger de le menche Longitud: 24
Array2: [e, n,  , e, n,  , l, e, g, e, r,  , d, e,  , l, e,  , m, e, n, c, h, e] Longitud: 24
Array3: [e, h, c, n, e, m,  , e, l,  , e, d,  , r, e, g, e, l,  , n, e,  , n, e] Longitud: 24
```

Clases `StringBuffer` y `StringBuilder`

Como hemos visto, la clase `String` nos facilita mucho el trabajo para manejar cadenas de caracteres. Sin embargo, los objetos `String` son constantes (inmutables), es decir, una vez que se crean su contenido nunca cambia; si, por ejemplo, se convierte a mayúsculas, en realidad se ha creado una nueva cadena, la original permanece intacta. Java dispone de la clase `StringBuffer`, que al igual que la clase `String`, representa una secuencia de caracteres, pero en este caso los objetos son modificables, es decir, las cadenas de caracteres pueden modificarse de forma dinámica.

Java distingue entre las cadenas constantes (inmutables) y las modificables por razones de rendimiento; en particular, Java puede realizar ciertas optimizaciones al manejar objetos `String`, como compartir una cadena constante entre varias referencias, y poder compararlas mediante los operadores `'=='` y `'!='` de forma más eficiente. Por eso, si sabemos que una cadena de caracteres no va a cambiar, conviene elegir la clase `String`; por otro lado, si vamos a realizar modificaciones sobre ella, entonces conviene elegir la clase `StringBuffer`.

Las tablas 4.6 y 4.7 resumen los constructores y los métodos de esta clase, respectivamente. Los métodos están sobrecargados (polimorfismo), por lo que para abreviar se muestra sólo un ejemplar de cada grupo de métodos sobrecargados. Fijándonos en la tabla de constructores, algunas formas de crear un objeto `StringBuffer` podrían ser las siguientes:

```
1 StringBuffer s1 = new StringBuffer ();        // Capacidad: 16
2 StringBuffer s2 = new StringBuffer ( 50 );    // Capacidad: 50
3 StringBuffer s3 = new StringBuffer ("abc");   // Capacidad: 16 + 3
```

Constructor	Descripción
`StringBuffer ()`	Construye un objeto sin caracteres y con una capacidad inicial de 16.
`StringBuffer (int cap)`	Construye un objeto sin caracteres y con la capacidad inicial especificada.
`StringBuffer (String s)`	Construye un objeto inicialziado con los caracteres de la cadena `String` especificada. Su capacidad inicial será de 16 más la longitud de la cadena `String`.

Cuadro 4.6: Clase `StringBuffer` - constructores

Constructor	Descripción
`append (...)`	Añade al final de la cadena de caracteres una representación en forma de cadena del valor recibido, que puede ser de cualquier tipo primitivo (`boolean`, `int`, ...) y objetos (incluyendo un `String` u otro `StringBuffer`). Este método es el que utiliza el compilador internamente para implementar los operadores `'+'` y `'+='` que concatenan objetos `String`.
`insert (int posicion, ...)`	Idem al anterior, pero en vez de añadir la cadena (el parámetro convertido a `String`) al final se inserta en la posición indicada.
`length ()`	Retorna el número de caracteres de la cadena actual (longitud).
`setLength ()`	Para modificar (aumentar o reducir) la longitud. Si la longitud especificada es menor que el número de caracteres que actualmente están en la cadena, ésta se truncará; si por el contrario, la longitud especificada es mayor, se anexarán caracteres nulos (`'\0'`).
`capacity ()`	Devuelve el número de caracteres que se pueden almacenar sin reservar más memoria.
`ensureCapacity (int capMin)`	Permite definir de forma manual la capacidad, para asegurarse de que se haya reservado un espacio suficiente para almacenar el número de caracteres especificado, sin esperar a que se ajuste automáticamente.
`trimToSize()`	Ajustar la capacidad al número exacto de caracteres, para reducir la memoria ocupada.
`charAt (int posicion)`	Obtener el carácter que está en la posición especificada.
`setCharAt (int pos, char c)`	Establecer un caracter en la posición especificada de la cadena.
`indexOf (String s [, pos])`	Busca la primera ocurrencia de una cadena y retorna su posición; si no la encuentra retorna -1.
`lastIndexOf (String s)`	Busca la última ocurrencia de una cadena y retorna su posición; si no la encuentra retorna -1.
`replace (int posIni, int posFin, String s)`	Sustituye los caracteres comprendidos entre las dos posiciones dadas por la cadena especificada.
`reverse ()`	Invierte el orden de los caracteres de la cadena actual.
`getChars ()`	Devuelve un array de caracteres, subcadena del `StringBuffer` en forma de un array de `char`.
`delete (int posIni, int posFin)`	Elimina los caracteres entre las posiciones indicadas de la cadena actual.
`deleteCharAt (int pos)`	Elimina el carácter de la posición indicada.
`substring (int posIni [, int posFin])`	Retornar una cadena `String` con los caracteres comprendidos entre el de la posición posIni, y el de la posición posFin, ambos inclusive.
`toString ()`	Retorna una representación en forma de cadena `String`.

Cuadro 4.7: Clase `StringBuffer` - métodos para manipulación de arrays

Desde la versión Java v1.5 (JDK 5), esta clase ha sido sustituida por la clase `StringBuilder`, la cual tiene exactamente las mismas operaciones (API) que `StringBuffer` pero es mucho más eficiente, ya que se ejecuta como un único hilo y no realiza operaciones de sincronización con otros hilos. Es decir, en caso de que necesitemos un objeto cadena de caracteres modificable, y éste vaya a ser utilizado desde diferentes hilos, entonces usaremos la clase `StringBuffer`; para el caso de que el objeto vaya a ser utilizado desde un único hilo (lo más común), entonces utilizaremos la clase `StringBuilder`.

Comparación de cadenas

Para comparar cadenas de caracteres hemos visto que la clase `String` ofrece una serie de métodos como el `equals()`, `equalsIgnoreCase()`, `compareTo()`, `startsWith()`, `endsWidth()`, y alguno más, los cuales comparan el contenido. Sin embargo, si dos cadenas se comparan mediante el operador `'=='` o `'!='`, como cuando se compara cualquier objeto, lo que se compara es la referencia, es decir, si realmente apuntan al mismo objeto. Podría darse el caso de que un objeto `String` tenga un valor, y se compare mediante `'=='` con una cadena de caracteres constante, y teniendo el mismo valor, el operador `'=='` retorne `false`. Consideremos el siguiente caso:

```
1  public class StringComparacion {
2      public static void main (String args[]) {
3          String s1 = "Hola";
4          String s2 = "Hola";
5
6          // Comparando s1 y s2
7          System.out.println ("Comparando con '==' ..... " + (s1 == s2));
8          System.out.println ("Comparando con equals() ... " + (s1.equals(s2)));
9
10         // Comparacion de s1 con la cadena "Hola"
11         System.out.println ("Comparando con '==' ..... " + (s1 == "Hola"));
12         System.out.println ("Comparando con equals() ... " + (s1.equals("Hola"))
               );
13
14         // Comparacion de s2 con la cadena "Hola"
15         System.out.println ("Comparando con '==' ..... " + (s2 == "Hola"));
16         System.out.println ("Comparando con equals() ... " + (s2.equals("Hola"))
               );
17     }
18 }
```

Observando la salida que genera parecería que no hay ningún problema en utilizar el operador `'=='` (o `'!='`), ya que en todos los casos, el resultado de la comparación es `true`, tanto cuando se comparan objetos como objetos con valores constantes, y tanto si se utiliza el operador `'=='` como el método `equals()`:

```
Comparando con '==' ..... true
Comparando con equals() ... true
Comparando con '==' ..... true
Comparando con equals() ... true
Comparando con '==' ..... true
Comparando con equals() ... true
```


Sin embargo, veamos el siguiente ejemplo:

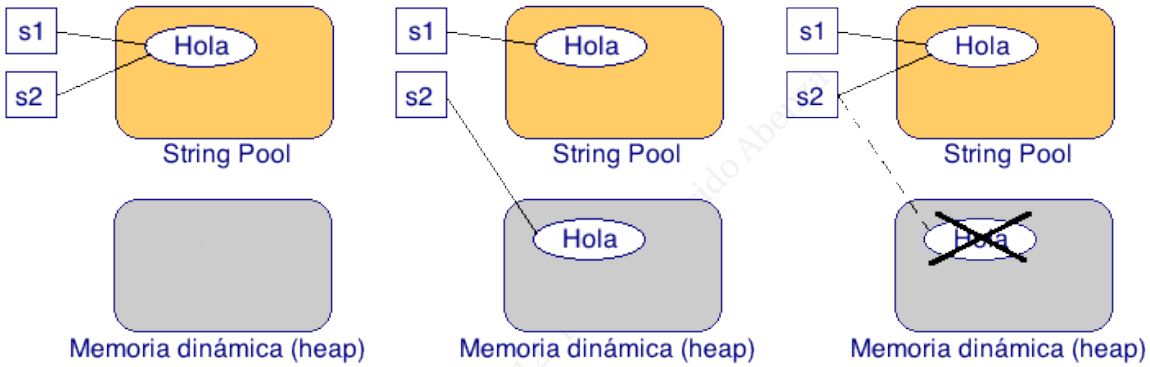
```
1  public class StringComparacion2 {
2     public static void main (String args[]) {
3        String s1 = "Hola";
4        String s2 = new String ("Hola");
5
6        // Repetiremos las comparaciones 2 veces
7        for (int i=1; i<=2; i++) {
8           // Comparando s1 y s2
9           System.out.println ("Comparando con '==' ..... " + (s1 == s2));
10          System.out.println ("Comparando con equals() ... " + (s1.equals(s2)));
11
12          // Comparacion de s1 con la cadena "Hola"
13          System.out.println ("Comparando con '==' ..... " + (s1 == "Hola"));
14          System.out.println ("Comparando con equals() ... " + (s1.equals("Hola"
                ")));
15
16          // Comparacion de s2 con la cadena "Hola"
17          System.out.println ("Comparando con '==' ..... " + (s2 == "Hola"));
18          System.out.println ("Comparando con equals() ... " + (s2.equals("Hola"
                ")));
19
20          // Internamos s2 (dinamico) y repetimos las comparaciones
21          s2 = s2.intern();
22          System.out.println ("");
23       }
24    }
25 }
```

En este caso, vemos que tenemos problemas al comparar con el operador `'=='` en el caso de que una de las cadenas a comparar se haya creado mediante el operador `new`, ya que la cadena se almacenará en la memoria dinámica, no en la zona de memoria para objetos `String`.

```
Comparando con '==' ..... false
Comparando con equals() ... true
Comparando con '==' ..... true
Comparando con equals() ... true
Comparando con '==' ..... false
Comparando con equals() ... true

Comparando con '==' ..... true
Comparando con equals() ... true
Comparando con '==' ..... true
Comparando con equals() ... true
Comparando con '==' ..... true
Comparando con equals() ... true
```


En Java, las cadenas de caracteres constantes (p.e. `"Hola"`) son objetos, pero en vez de almacenarse en la memoria dinámica se almacenan en una zona específica de la memoria llamada "area de memoria de objetos `String`" o *String Pool*. Esta zona consiste en algo similar a un diccionario de palabras, donde no pueden haber dos palabras iguales. Por tanto, si en un mismo programa se escribe dos o más veces una misma cadena de caracteres, el compilador no añade una nueva entrada sino que utiliza la que ya teníamos, es decir, lo referencia, ahorrando memoria. Los objetos `String` creados utilizando el operador `new` no utilizan esa zona, sino que se reserva espacio para él en la memoria dinámica o *heap*. Por ello, si comparamos objetos `String` utilizando el operador `'=='` o `'!='` para comparar cadenas de caracteres no funciona cuando se trata de comparar objetos `String` creados mediante `new`, ya que la dirección de memoria será diferente. Sin embargo, podemos "internar" la cadena de caracteres que encapsula dicho objeto `String`, ejecutando el método `intern()`, el cual busca si existe ya una cadena así en el area de memoria de objetos `String`, y si no, la añade; en cualquier caso, retorna una referencia a dicha cadena ya insertada.




```
String s1 = "Hola";
String s2 = "Hola";
```

```
String s1 = "Hola";
String s2 = new
   String ("Hola");
```

```
String s1 = "Hola";
String s2 = new
   String ("Hola");
s2 = s2.intern ();
```

El motivo de este mecanismo es doble:

1. **Ahorrar memoria**, al no tener dulicadas cadenas de caracteres constantes (literales).
2. **Mejorar el rendimiento**, pues podemos comparar cadenas utilizando el operador `'=='` o `'!='`, lo cual es más rápido que utilizar el método `equals()` o `equalsIgnoreCase()`, ya que simplemente se compara dos referencias (direcciones), no su contenido; comparar el contenido de dos cadenas requiere una comparación de cada uno de los pares de caracteres de la misma posición en cada `String`.

La **conclusión** que podemos extraer a la vista de estos ejemplos es la de no comparar nunca cadenas de caracteres dinámicas (creadas mediante `new`) utilizando el operador `'=='`. Por tanto, para evitar este problema podemos seguir uno o los dos siguientes consejos:

- No comparar nunca dos cadenas de caracteres mediante los operadores `'=='` o `'!='`, utilizar siempre los métodos `equals()`, `equalsIgnoreCase()`, o cualquier otro de los que ofrece la clase `String`. El entorno de desarrollo NetBeans nos avisa de esta circunstancia, aunque no quiere decir que sea un error. En caso de querer comparar si dos cadenas no son iguales, utilizar el operador de negación `'!'` delante de la invocación al método (o comparar con `false`). Lo siguiente son dos posibles formas de comparar dos cadenas de caracteres `s1` y `s2` de forma segura, tanto para comparar la igualdad como para comparar la desigualdad:

```
// Igualdad
if (s1.equals(s2)) {...}
```

```
// Desigualdad
if (!s1.equals(s2)) {...}
```

```
// Igualdad
if (s1.equals(s2)==true)) {...}
```

```
// Desigualdad
if (s1.equals(s2)==false)) {...}
```

- Evitar la creación de objetos `String` mediante el operador `new`, esto es, haciendo uso de alguno de sus constructores. El entorno de desarrollo NetBeans nos avisa de esto, ofreciendo la posibilidad de corregirlo automáticamente. En caso de que se creen los `String` mediante el operador `new`, internar la cadena en el área de memoria de objetos `String` mediante el método `intern()`, actualizando la referencia a dicha cadena:

```
// Igualdad
String s1 = "...";
String s2 = new String (...);
...
s2 = s2.intern();
if (s1==s2) {...}
```

```
// Desigualdad
String s1 = "...";
String s2 = new String (...);
...
s2 = s2.intern();
if (s1!=s2) {...}
```


Concatenación de cadenas

En cuanto a **concatenar cadenas de caracteres**, en algún ejemplo ya se ha utilizado el operador `'+'`, que también podría ser el `'+='`. Hemos visto que también admite concatenar cualquier valor numérico, ya que se convierte automáticamente a cadena en el momento en que uno de los dos operandos del operador `'+'` sea un `String`. Además, la clase `String` dispone del método `concat()`, como se presentó en la tabla 4.5. El siguiente ejemplo utiliza todas estas posibilidades de concatenación:

```
1  public class StringManipulacion3 {
2     public static void main (String args[]) {
3        String palabras[] = { "En", "un", "lugar",
4                              "de", "la", "mancha" };
5        String frase;
6
7        // Concatenamos las palabras del array de String
8        frase = "";
9        for (int i=0; i<palabras.length; i++) {
10          if (i>0) frase = frase + " ";
11          frase = frase.concat (palabras[i]);
12       }
13       // Concatenamos el numero de palabras
14       frase += " - Numero palabras: " + palabras.length;
15
16       // Mostramos la cadena final con la frase completa
17       System.out.println ("Cadena final: " + frase);
18    }
19 }
```

Generando la salida siguiente:

```
Cadena final: En un lugar de la mancha - Numero palabras: 6
```

Formateo de cadenas

La forma más sencilla de componer una cadena, bien sea para imprimirla o almacenarla es utilizando el operador '+', tal cual se ha explicado en la sección anterior.

```
// Imprimir
int n = 5;
System.out.println ("Valor: " + n);
```

```
// Almacenar
int n = 5;
String s = "Valor: " + n;
System.out.println (s);
```

Otro mecanismo alternativo heredado del lenguaje C sería utilizar el método estático `format()`. Los siguientes ejemplos muestran dos fragmentos de código equivalentes a los anteriores:

```
// Imprimir
int n = 5;
System.out.printf ("Valor: %d", n);
```

```
// Almacenar
int n = 5;
String s = String.format
              ("Valor: %d", n);
System.out.println (s);
```

Se puede apreciar el uso de la función `printf()` y `format()` con "máscaras" para el formateo similares a los del lenguaje C. La tabla 4.8 muestra algunas de las más habituales; muchas de ellas admiten variaciones, para especificar el número de decimales, relleno de ceros, etc. Para consultar exactamente las máscaras disponibles consultar la documentación en línea de la librería de clases de Java (API Specification), en concreto, la función `format()` de la clase `String`, o directamente, la clase `Formatter`.

Máscara(s)	Descripción
%d	El resultado será un valor entero en decimal.
%o	El resultado será un valor entero en octal.
%x %X	El resultado será un valor entero en hexadecimal.
%f %g	El resultado será un valor en real (con decimales).
%s	El resultado será una cadena de texto.

Cuadro 4.8: Máscaras de formateo

Una posibilidad para poder establecer un **número máximo de decimales** de los valores reales (`float` o `double`) sería con la clase `DecimalFormat`, del paquete `java.text`. Tan sólo se muestra el siguiente ejemplo, el cual imprime por pantalla el valor de PI (`double`) con 3 decimales, redondeando su verdadero valor. Simplemente explicar que a la hora de crear el objeto de la clase `DecimalFormat` se especifica una máscara, que será la que se utilizará después para formatear la salida a mostrar por pantalla. En este caso vemos que la máscara incluye símbolos de almohadillas '#' y '0'; las almohadillas significan valores que se rellenarán si el valor llega a necesitarlos, mientras que los valores '0' significan que siempre saldrán, aunque no sean necesarios. Por tanto, con esa máscara estamos limitando el número de decimales a 3: si el valor a formatear tiene más decimales se redondeará, si tiene menos (o no tiene decimales), se rellenarán con ceros para tener el número de decimales requerido.


```java
import java.text.DecimalFormat;

public class DecimalFormatDemo {
   public static void main (String args[]) {
      // Mascara para tres decimales
      DecimalFormat f = new DecimalFormat ("#,##0.000");

      // Imprimimos el valor de PI
      System.out.println ("PI: " + Math.PI);
      System.out.println ("PI: " + f.format(Math.PI));
   }
}
```

Vemos que obtenemos la salida deseada, aunque el símbolo utilizado para los decimales es distinto en los dos casos: cuando se imprime un valor tal cual se utiliza el punto '.', mientras que al utilizar la máscara nos sale con la coma ',', que es el símbolo definido en nuestro sistema.

```
PI: 3.141592653589793
PI: 3,142
```

Existen varias soluciones, por ejemplo, la siguiente, que establece manualmente los símbolos utilizados para coma decimal y separador de miles; otras soluciones consisten en utilizar `Locale`.

```java
import java.text.*;

public class DecimalFormatDemo {
   public static void main (String args[]) {
      // Mascara para tres decimales
      DecimalFormat f = new DecimalFormat ("#,##0.000");

      // Imprimimos el valor de PI
      System.out.println ("PI: " + Math.PI);

      DecimalFormatSymbols symbols = f.getDecimalFormatSymbols();
      symbols.setDecimalSeparator ('.');
      symbols.setGroupingSeparator(',');
      f.setDecimalFormatSymbols(symbols);
      System.out.println ("PI: " + f.format(Math.PI));
   }
}
```

```
PI: 3.141592653589793
PI: 3.142
```


División de cadenas

En ocasiones puede que necesitemos dividir una cadena de caracteres `String` según algún carácter delimitador (espacios en blanco, tabuladores, saltos de línea, retornos de carro, comas, etc.), y obtener diferentes subcadenas (*tokens*). Por ejemplo, la división de una frase en palabras, donde el separador será el espacio en blanco. La acción de extraer una secuencia de *tokens* a partir de la cadena recibe la denominación de análisis lexicográfico. Para esta tarea podemos utilizar lo siguiente:

- Clase `StringTokenizer`
- Método `split()` de la clase `String`
- Clase `Scanner`

Clase `StringTokenizer`

La clase `StringTokenizer` del paquete `java.util` se utiliza para separar un cadena `String` en partes llamadas *tokens*. Se puede especificar una cadena con los carateres utilizados como delimitador, aunque si no se especifica se utilizarán los más habituales: espacio en blanco, salto de línea, tabulador, etc. (ver el constructor por defecto en la tabla 4.9).

Constructor	Descripción
`StringTokenizer (String s)`	Construye un objeto para la cadena especificada, separándola con cualquiera de los deliminadores por defecto: espacio en blanco, '\t', '\n', '\r', y '\f'.
`StringTokenizer (String s, String delim)`	Construye un objeto para la cadena especificada, utilizando cualquiera de los deliminadores especificados para separarla.
`StringTokenizer (String s, String delim, boolean incluirDelim)`	Idem al anterior, pero incluyendo (`true`) o no (`false`) los delimitadores también como *tokens* (de longitud 1).

Cuadro 4.9: Clase `StringTokenizer` - constructores

Una vez creado el objeto `StringTokenizer` con la cadena de la que queremos extraer las unidades lexicográficas, a continuación podemos ir extrayendo tokens con los métodos de la tabla 4.10.

Constructor	Descripción
`countTokens()`	Retorna el número de tokens pendientes de analizar.
`hasMoreTokens()`	Averiguar si quedan más tokens por analizar.
`nextToken()`	Nos retorna el próximo *token* (`String`), y avanza hasta el siguiente. Si ya no quedan más *tokens*, nos genera la excepción `NoSuchElementException`.

Cuadro 4.10: Clase `StringTokenizer` - métodos

El siguiente ejemplo separa en palabras una frase, utilizando como carácter separador el espacio en blanco. Nos muestra el número de palabras (*tokens*) y cada una de ellas en una línea independiente, como podemos comprobar ejecutando el programa:


```java
import java.util.*;

public class StringTokenizerDemo {
   public static void main (String args[]) {
      String s = "En un lugar de La Mancha";
      int numPalabras = 0;

      // Separamos la cadena
      //    (el espacio en blanco como delimitador)
      StringTokenizer tokens = new StringTokenizer (s, " ");
      // Imprimimos resultado
      numPalabras = tokens.countTokens();
      System.out.println ("Número de palabras: " + numPalabras);
      while ( tokens.hasMoreTokens() ) {
         System.out.println( tokens.nextToken() );
      }
   }
}
```

```
Número de palabras: 6
En
un
lugar
de
La
Mancha
```

La clase `StringTokenizer` puede generar resultados inesperados en ciertas situaciones, como cuando en la cadena a analizar se encuentra dos caracteres separadores juntos. Por ejemplo, si utilizamos la coma como separador de campos, la cadena ‘‘a,,b’’ nos retorna sólo 2 campos en lugar de 3, ignorando el campo vacío del centro. Modificando la cadena en el programa anterior, y el carácter separador a ‘‘,’’, el resultado del programa es el siguiente:

```
Número de palabras: 2
a
b
```

No se aconseja utilizar la clase `StringTokenizer` para nuevos desarrollos, ya que se mantiene por compatibilidad con versiones anteriores. Se recomienda utilizar mejor el método `split()` que se explica a continuación, el cual sí soluciona el problema que acabamos de comentar.

Método `split()` de la clase `String`

El método `split()` utiliza expresiones regulares como separadores, es decir, no sólo un carácter o varios expresados como una cadena, sino también expresiones "patrones" (máscaras, operadores lógicos, rangos de caracteres, etc.). El uso de expresiones regulares se explica en la documentación en línea, concretamente en la clase `Pattern`.

Lo siguiente es un ejemplo de uso de este método realiza la misma tarea que el programa que utilizaba la clase `StringTokenizer`, como puede observarse en el resultado de su ejecución:

```
1  public class StringSplitDemo {
2     public static void main (String args[]) {
3        String s = "En un lugar de La Mancha";
4        String[] palabras;
5        int numPalabras = 0;
6
7        // Separamos la cadena
8        //    (el espacio en blanco como delimitador es '\s')
9        palabras = s.split ("\\s");
10       numPalabras = palabras.length;
11       // Imprimimos resultado
12       System.out.println ("Número de palabras: " + numPalabras);
13       for (int i=0; i<numPalabras; i++) {
14          System.out.println (palabras[i]);
15       }
16    }
17 }
```

```
Número de palabras: 6
En
un
lugar
de
La
Mancha
```

En el segundo ejemplo presentado con la clase `StringTokenizer` vimos que no se contaba bien el número de *tokens*. Utilizando la misma cadena aquel ejemplo y la coma como carácter separador, podemos comprobar que el método `split()` sí tiene en cuenta los campos vacíos cuando hay dos separadores juntos.

```
1  public class StringSplitDemo2 {
2     public static void main (String args[]) {
3        String s = "a,,b";
4        String[] palabras;
5        int numPalabras = 0;
6
7        // Separamos la cadena
8        //    (la coma como delimitador es ',')
```



```
9         palabras = s.split (",");
10        numPalabras = palabras.length;
11        // Imprimimos resultado
12        System.out.println ("Número de palabras: " + numPalabras);
13        for (int i=0; i<numPalabras; i++) {
14           System.out.println (palabras[i]);
15        }
16     }
17  }
```

```
Número de palabras: 3
a

b
```

Vamos a mostrar dos ejemplos más, y para reducir un poco el código imprimiremos directamente los arrays resultantes en vez de realizar un bucle.

```
1  import java.util.Arrays;
2
3  public class StringSplitDemo3 {
4     public static void main (String args[]) {
5        String s = "a, b, c,d";  // Se han juntado c y d
6        String[] palabras;
7        int numPalabras = 0;
8
9        // Separamos la cadena (separador es , y espacio en blanco)
10       palabras = s.split (", ");
11       numPalabras = palabras.length;
12       // Imprimimos resultado
13       System.out.println ("Número de palabras: " + numPalabras + " -> " +
             Arrays.toString(palabras));
14    }
15 }
```

```
Número de palabras: 3 -> [a, b, c,d]
```

```
1  import java.util.Arrays;
2
3  public class StringSplitDemo4 {
4     public static void main (String args[]) {
5        String s = "Pedro, Ana, and Pepe";
6        String[] palabras;
7        int numPalabras = 0;
8
```



```
9         // Separamos la cadena (separador es expresion regular)
10        palabras = s.split ("[, ]+(and|or)*[, ]*");
11        numPalabras = palabras.length;
12        // Imprimimos resultado
13        System.out.println ("Número de palabras: " + numPalabras + " -> " +
              Arrays.toString(palabras));
14    }
15 }
```

```
Número de palabras: 3 -> [Pedro, Ana, Pepe]
```

Para finalizar, como ejemplo de uso de la clase `StringBuilder` (versión optimizada de `StringBuffer`) vamos a presentar un programa que invierte el orden de las palabras de una cadena String. Se puede observar que para todos los pasos intermedios se utiliza un objeto `StringBuilder`, y al final vuelve a convertirse a un `String`; esto es mucho más eficiente que utilizar únicamente `String`, aunque también podría utilizarse.

```
1  public class StringSplitDemo5 {
2     public static void main (String args[]) {
3        String s1 = "En un lugar de La Mancha";
4        String[] palabras = s1.split ("\\s");
5        StringBuilder sb = new StringBuilder();
6        for (int i=0; i<palabras.length; i++) {
7           sb.insert (0, palabras[i]);
8           if (i < palabras.length - 1) {
9              sb.insert(0, " ");
10          }
11       }
12       String s2 = sb.toString();
13       // Mostramos la cadena original y la final
14       System.out.println (s1);
15       System.out.println (s2);
16    }
17 }
```

```
En un lugar de La Mancha
Mancha La de lugar un En
```


Clase `Scanner`

La clase `Scanner` del paquete `java.util` nos permite realizar tareas de separación de cadenas, de forma muy parecida a las vistas con la clase `StringBuffer`, `StringBuilder`, o el método `split()`.

```
1  import java.util.Scanner;
2
3  public class ScannerDemo {
4     public static void main (String args[]) {
5        String str = "En un lugar de La Mancha";
6        Scanner sc = new Scanner(str).useDelimiter("\\s");
7        while (sc.hasNext()) {
8           System.out.println (sc.next());
9        }
10       sc.close();
11    }
12 }
```

```
En
un
lugar
de
La
Mancha
```

El siguiente ejemplo utiliza una expresión regular algo más compleja, utiliza la palabra “globos” como separador, rodeada de cualquier número de espacios en blanco. Lo que queda son los valores numéricos, que los procesamos como valores enteros con `nextInt()`.

```
1  import java.util.Scanner;
2
3  public class ScannerDemo2 {
4     public static void main (String args[]) {
5        String str = "1   globos  2 globos 3   globos";
6        Scanner sc = new Scanner(str).useDelimiter("\\s*globos\\s*");
7        while (sc.hasNext()) {
8           System.out.println (sc.nextInt());
9        }
10       sc.close();
11    }
12 }
```

```
1
2
3
```


Para terminar, comentar que la clase `Scanner` se utiliza también para leer datos por teclado desde la consola, utilizando el flujo de entrada estándar (`System.in`) como entrada en vez de una cadena `String` como en los ejemplos anteriores. Lo siguiente muestra únicamente el fragmento de código que sería necesario para introducir un valor entero por teclado:

```
1 Scanner sc = new Scanner(System.in);
2 int i = sc.nextInt();
```

GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere