Capítulo 8. Cadenas de caracteres

A. J. Pérez

Cadenas de caracteres

Una cadena o `String` es una secuencia de caracteres almacenados en posiciones consecutivas de la memoria; es equivalente a un vector de caracteres de una determinada longitud. Las cadenas de caracteres son parte fundamental de la mayoría de los programas y tienen consideración particular -como tipo de dato- en los lenguajes de programación.

En Java las cadenas de caracteres son objetos con características especiales que debido a su uso frecuente **admiten sintaxis y usos semejantes a los tipos primitivos** del lenguaje. Cada carácter dentro de un **`String`** se codifica empleando un *Formato de Transformación Unicode* UTF-16.

La clase `java.lang.`**`String`** encapsula un vector (array) de caracteres junto a la funcionalidad necesaria para su manejo.

La clase **`String`** viene marcada, en su código fuente, con el modificador **`final`** para que no sea posible extender ni reescribir la clase.

String inmutables

Los objetos de la clase `String` son *inmutables*; significa que **una vez creados en memoria no pueden ser alterados ni se puede cambiar su valor**; sin embargo **como referencias, sí pueden reasignarse para apuntar a otro objeto**.

Un ejemplo para entender esta particularidad es:

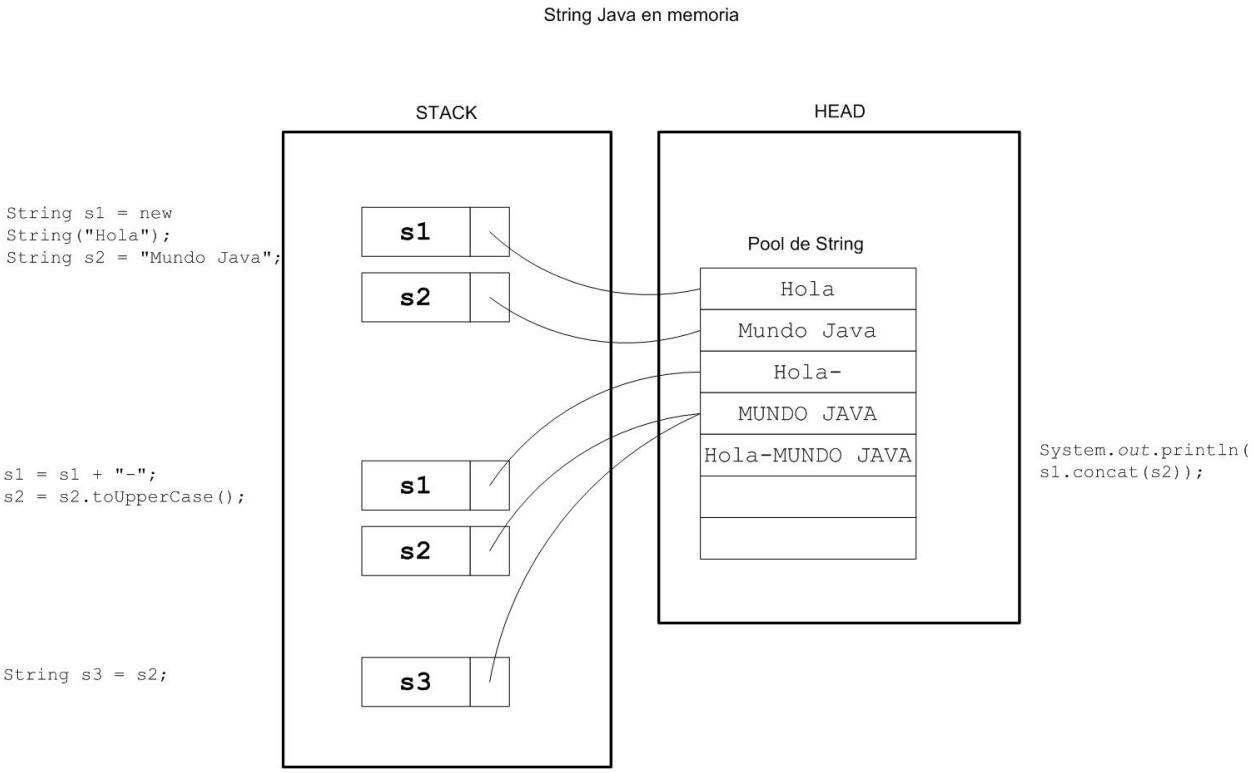
```
public class ObjetoInmutable {
	static public void main(String[] args) {
	String s1 = new String("Hola");
	String s2 = "Mundo Java";

	System.out.println(s1);
	System.out.println(s2);

	s1 = s1 + "-";
	s2 = s2.toUpperCase();                  // Pasa a mayúsculas

	System.out.println(s1);
	System.out.println(s2);
	System.out.println(s1.concat(s2));      // Concatena dos cadenas.
                                            // Equivale a s1 + s2
	String s3 = s2;
```

```
        }
}
```

Salida:

```
Hola
Mundo Java
Hola-
MUNDO JAVA
Hola-MUNDO JAVA
```

La representación en la memoria del ejemplo anterior sería algo así:



El almacenamiento está controlado por la *JVM*. Cuando se inicializan las variables, se les asignan los valores `"Hola"` y `"Mundo Java"`; es cuando se crean los dos primeros objetos de tipo **`String`**. Después, a `s1` se concatena `"-"` y se guarda el resultado en la misma variable, haciendo que se genere un nuevo objeto, actualizando la referencia para que apunte a éste. A `s2` se le transforma en mayúsculas; nuevamente se genera otro objeto, asignando su referencia en `s2`. Y por último, se produce una salida en pantalla de `s1 + s2`. En este caso, también se genera un nuevo objeto, con la particularidad de que no se asocia a ninguna variable; una vez que se muestra por pantalla el texto el

objeto asociado es candidato a ser eliminado por el *garbage collector*. Como se puede comprobar, **en ningún momento se modifica ningún objeto que estuviera en memoria, pero sí se cambian las referencias asignadas en `s1` y `s2`**. Por último, **cuando se crea un nuevo objeto `s3`, y se le asigna `s2`, no se crea un nuevo objeto, se asigna la misma posición de memoria**.

Cada vez que define una nueva cadena, se genera un nuevo objeto **`String`**. La única excepción es cuando se asigna el valor de una variable **`String`** a otra. En este caso, las dos apuntan al mismo objeto en memoria.

String y la memoria

Dado que el uso de cadenas de caracteres es algo muy común, la *JVM* reserva un área especial de la memoria denominada ***`String`*** *constant pool* (banco de literales **`String`**) para optimizar la memoria. Cuando el compilador encuentra un literal de tipo **`String`**, comprueba en el *banco* si existe otra cadena idéntica. Si se encuentra una, directamente se referencia a ésta, sino, se crea un nuevo objeto **`String`**.

Hay dos clases que acompañan a **`String`**, llamadas **`StringBuilder`** y **`StringBuffer`**, que se utilizan para crear cadenas que pueden ser alteradas después de ser creadas.

Creación de String y constructores

Existe una diferencia sutil entre crear un **`String`** mediante un literal y un objeto construido, como todos los demás, a partir de clases normales con el operador **`new`**.

Por ejemplo, con:

```
String s = "Mundo Java";
```

Se crea una variable de referencia `s` y un objeto **`String`** que se almacena en el *banco de literales String* quedando vinculado a `s`.

Sin embargo con:

```
char caracteres[] = {'M','u','n','d','o',' ','J','a','v','a'};
String s = new String(caracteres);         // s es la cadena "Mundo Java"
```

Se crea una variable de referencia s que, como se ha utilizado el operador **`new`**, crea un objeto **`String`** en el *Heap,* fuera del *banco de literales String,* quedando vinculado a `s`.

Este ejemplo:

```
String s = new String();
```

Crea una instancia de **`String`** sin caracteres en ella, `""`. No es lo mismo que **`null`**.

Si se tiene un *vector* de caracteres del que sólo nos interesa una parte, existe un constructor que permite especificar el *índice* de comienzo y el *número* de caracteres a utilizar.

```
char caracteres[] = {'M','u','d','o',' ','J','a','v','a'};
String s = new String(caracteres, 5, 4);    // s es la cadena "Java"
```

También existen constructores para caracteres *ASCII* (caracteres de 8 bits) frente a los caracteres *Unicode* de Java (caracteres de 16 bits).

Sintaxis especial

Java incluye algunas ayudas sintácticas con el fin facilitar y simplificar las operaciones más habituales con cadenas.

Llamadas a métodos con String literales

Una de estas facilidades sintácticas de **`String`** se manifiesta cuando se llama a los métodos directamente con una cadena literal entre comillas, tal como si fuera una referencia a objeto.

Ejemplo:

```
String s = "abc";
System.out.println("abc".lenght());    // muestra 3
```

Concatenación de cadenas

El operador que utiliza Java con los objetos **`String`** es `+`. El **+** actúa como **operador de concatenación**, en este caso en concreto para mejorar la legibilidad, por ser una operación muy habitual. **El operador + realmente no añade; genera nuevos objetos con la concatenación**, debido a la naturaleza *inmutable* de los objetos **`String`.**

```
String s1 = "Juan tiene " + edad + " años";
```

Está más claro que…

```
String s2 = new StringBuilder("Juan tiene ").
                            append(edad).append(" años").toString();
```

El método `append()` añade caracteres al final del **`StringBuilder`**, y `toString()` convierte en **`String`** el **`StringBuilder`**. Más adelante se tratará `append()` y `toString()` con más detalle.

Precedencia de operadores

Cuando se combinan expresiones numéricas enteras con expresiones de concatenación de cadenas hay que tener cuidado porque se pueden obtener resultados no esperados:

```
String s = "cuatro: " + 2 + 2;
```

Se podría esperar que el valor de `s` sea **`"cuatro: 4"`**, pero la precedencia de operadores provoca que se evalúe primero la subexpresión `"cuatro: " + 2`, y después `"cuatro: 2" + 2`, donde el resultado es **`"cuatro: 22"`**. Si se desea realizar primero la expresión entera, hay que utilizar paréntesis, como aquí:

```
String s = "cuatro: " + (2+2);
```

Conversión de cadenas

`StringBuilder` y **`StringBuffer`** tienen una versión sobrecargada de `append()` para cada tipo posible. Por lo tanto, cuando se utiliza **+** para concatenar una variable, se llama a la versión adecuada de `append()` para esa variable. El método `append()` realmente llama a un método estático de **`String`** llamado `valueOf()` para construir la representación de tipo **`String`**. Para tipos primitivos, `valueOf()` crea simplemente una representación de cada **`int`** o **`float`**. Para objetos, `valueOf()` llama al método `toString()` del objeto. Todas las clases disponen de un método `toString()` por defecto que se encuentra en la clase **`Object`**. Es conveniente redefinir `toString()` y presentar una versión propia de cómo sería un objeto pasado a texto en todas las clases que se programan.

El ejemplo siguiente muestra una clase que sobrescribe `toString()` para mostrar los valores de sus atributos.

```
public class Punto {
        int x, y;

        //Redefinición del método
```

```
	public String toString() {
		return "Punto[" + x + "," + y + "]";
	}
}

class toStringDemo {
	public static void main(String args[]) {
		Punto p = new Punto();		//por defecto 0,0
		System.out.println(p);		//Punto[0,0]
	}
}
```

Esta versión de la clase **Punto** incluye la redefinición del método `toString()` de **Object**. Da formato a la cadena que contiene los valores de `x` e `y` de cada instancia de **Punto**.

Salida:

```
Punto[0,0]
```

Resumen de métodos de la clase String

A continuación se describen los métodos más utilizados de la clase **String**:

Tipo de retorno	Método	Descripción
char	`charAt(int indice)`	Devuelve el carácter ubicado en el índice especificado.
String	`concat(String str)`	Agrega un **String** al final de otro (equivalente a utilizar el +). Produce un nuevo **String** con el resultado.
int	`compareTo(String otroString)`	Devuelve *menor* que **0** si la cadena que se compara es menor que `otroString`, **0** si son iguales y *mayor* que **0** si la cadena es mayor que `otroString`. Tiene en cuenta las mayúsculas.
boolean	`equals(String otroString)`	Indica si dos cadenas son iguales, tiene en cuenta las mayúsculas.
boolean	`equalsIgnoreCase(String otroString)`	Indica si dos cadenas son iguales, sin tener en cuenta mayúsculas.

void	`getChars(int i, int j, char[] destino, int p)`	Copia los caracteres indicado entre los índices `i` y `j`. Los pone en la posición p del array `destino`.
int	`indexOf(String subString)`	Devuelve la posición donde empieza una subcadena dentro de la cadena.
int	`length()`	Devuelve la cantidad de caracteres que contiene la cadena.
String[]	`split(String regex)`	Recibe como parámetro una expresión regular. Devuelve un array con los *tokens* que se obtienen al cortar según por el *parser* especificado con `regex`.
String	`replaceAll(String regex, String textoReemplazo)`	Reemplaza, en un nuevo **`String`**, todas las coincidencias de `regex` con el texto de reemplazo.
String	`replace(char caracterViejo, char caracterNuevo)`	Reemplaza, en un nuevo **`String`**, cada `caracterViejo` con el `caracterNuevo`.
boolean	`regionMatches(int tof, String s, int oof, int l)`	Comprueba si coinciden dos subcadenas de longitud `l`; una empieza en `tof`, otra empieza en `oof` de la cadena `s`.
String	`subString(int inicioIndice, int finIndice)`	Devuelve una nueva cadena con los caracteres comprendidos desde `inicioIndice` hasta `finIndice`.
char[]	`toCharArray()`	Convierte un **`String`** en un array de **`char`**.
String	`toLowerCase()`	Sobre un nuevo **`String`**, convierte los caracteres en minúsculas.
String	`toString()`	Devuelve el texto de un **`String`** (dado que es un **`String`**, se devuelve a sí mismo).
String	`toUpperCase()`	Sobre un nuevo **`String`**, convierte los caracteres en mayúsculas.
String	`trim()`	Sobre un nuevo **`String`**, elimina los espacios en blanco al comienzo y final de la cadena.
String	`valueOf(…)`	Devuelve el texto resultado de convertir el valor de un tipo primitivo cualquiera recibido como argumento.

Extracción de caracteres

Para extraer un único carácter de una cadena, se utiliza `charAt()` indicando entre paréntesis el índice –empezando por `0`- de un carácter de la cadena:

```
"abc".charAt(1) // devolverá 'b'
```

Si se necesita extraer más de un carácter a la vez, puede utilizar el método `getChars()`, que permite especificar el índice del primer carácter y del último más uno que se desean copiar, además de un *array* **`char`** donde se desean colocar dichos caracteres.

```
String texto = "Esto no es una canción";
char buf[] = new char[2];
texto.getChars(5, 7, buf, 0); // buf ahora tendrá el valor "no"
```

También existe un método útil, llamado `toCharArray()`, que devuelve un *array* de **`char`** que contiene la cadena completa.

Comparación

Si se desean comparar dos cadenas para ver si son iguales, puede utilizarse el método `equals()` de **`String`**. Devolverá **`true`** si el único parámetro está compuesto de los mismos caracteres que el objeto con el que se llama a `equals()`. Una forma alternativa de `equals()` llamada `equalsIgnoreCase()` ignora si los caracteres de las cadenas que se comparan están en mayúsculas o minúsculas.

La clase **`String`** ofrece un par de métodos útiles; versiones especializadas de `equals()`. El método `regionMatches()` que se utiliza para comparar una parte específica de una cadena, con otra parte de otra cadena. Hay dos variantes de `regionMatches()`, una permite controlar si es importante la diferenciación entre mayúsculas/minúsculas; la otra asume que sí lo es.

```
// Distingue las mayúsculas
boolean regionMatches(int toffset, String otra, int ooffset, int longitud);

 // No distingue las mayúsculas
boolean regionMatches(boolean ignorarMaysc,  int toffset, String otra,  int offset,
int longitud);
```

En estas dos versiones de `regionMatches()`, el parámetro `toffset` indica el desplazamiento en

caracteres en el objeto **`String`** sobre el que estamos llamando el método. La cadena con la que estamos comparando se llama `otra`, y el desplazamiento dentro de esa cadena se llama `ooffset`. Se comparan las subcadenas de longitud especificada de las dos cadenas comenzando a partir de los dos desplazamientos.

Igualdad

El método `equals()` y el operador == hacen dos pruebas completamente diferentes para la igualdad. Mientras que el método `equals()` compara los caracteres contenidos en un **`String`**, el operador == compara dos referencias a objeto para ver si se refieren a la misma instancia.

Ordenación

A menudo no basta con conocer si dos cadenas son idénticas o no. Para aplicaciones de ordenación, es necesario conocer cuál es *menor que*, *igual que* o *mayor que*. El método `compareTo()` se puede utilizar para determinar la posición en un proceso ordenación. Si el resultado entero de `compareTo()` es negativo, la cadena -que ejecuta el método- es menor que el parámetro, y si es positivo, la cadena es mayor. Si `compareTo()` devuelve **`0`**, entonces las dos cadenas son iguales.

En el siguiente ejemplo se ordena un *vector* de cadenas utilizando `compareTo()`. Utiliza el algoritmo de *ordenación de la burbuja*.

```
class OrdenStringBurbuja {
	static String vector[] = {
		"Ahora", "es", "el ", "momento", "de", "actuar"
	};

	public static void main(String args[])  {
		for (int j = 0; j < vector.length; j++) {
			for (int i = j + 1; i < vector.length; i++) {
				if (vector[i].compareTo(vector[j]) < 0) {
					String aux = vector[j];
					vector[j] = vector[i];
					vector[i] = aux;
				}
			}
			System.out.println(vector[j]);
		}
	}
}
```

valueOf()

Si se tiene algún tipo de datos y se desea mostrar su valor con algún formato de texto, primero hay que

convertirlo a **`String`**. El método `valueOf()` está sobrecargado en todos los tipos posibles de Java, por lo que cada tipo se puede convertir correctamente en un `String`. Cualquier objeto al que se le pida `valueOf()` devolverá el resultado de llamar al método `toString()` del objeto. De hecho, se podría llamar directamente a `toString()` y obtener el mismo resultado.

StringBuilder y StringBuffer

Como se ha visto anteriormente, las cadenas en Java son *inmutables*. Esto significa que todas las modificaciones realizadas sobre un objeto **`String`**, se aplican realmente sobre otro nuevo que contiene los cambios. Por ejemplo, utilizando los métodos `replace()` , `toUpperCase()` , `trim()`, etc. no cambian el objeto **`String`** para el que fueron llamados, y utilizan un nuevo bloque de memoria en el que se almacena el nuevo contenido. Esto tiene ventajas en los usos habituales, pero en algunos casos puede causar problemas de rendimiento en las aplicaciones, si no se tiene en cuenta. Un problema similar se produce cuando se unen **`String`** en un bucle, independientemente de si se realiza la concatenación con el método `concat()` o con los operadores **+** y **+=** . El problema está directamente relacionado con el procesamiento de objetos **`String`** y de la memoria dinámica en la que se almacenan.

Para comprender el procedimiento de concatenación de cadenas en Java, por ejemplo, tomemos dos variables `s1` y `s2` del tipo **`String`** , con valores de `"Hola"` y `"Mundo Java"`. En el *heap* de memoria dinámica, hay dos bloques donde se almacenan los valores. La finalidad de `texto1` y `texto2` es mantener una referencia a una dirección de memoria donde se almacena los datos. La variable `resultado` recibe la concatenación de las dos cadenas.

Un fragmento de código para crear y definir las tres variables podría ser:

```
String  texto1 = "Hola" ;
String  texto2 = "Mundo Java" ;
String resultado = texto1 + texto2;
```

Al crearse la variable resultado se ubica en un nuevo bloque de memoria en el *Heap*, que almacena el valor `"HolaMundo Java"`. La variable `resultado` tendrá asignada la nueva dirección de memoria. Como resultado de ello, tendremos tres áreas de memoria y tres referencias. **La creación de una nueva cadena y hacer referencia a su bloque de memoria es un proceso que lleva tiempo que acaba siendo un problema en un bucle que se repite muchas veces**.

A diferencia de otros lenguajes de programación en Java no se define un *destructor*, es decir, No hay necesidad de liberar manualmente los objetos almacenados en la memoria. Existe un mecanismo especial denominado *recolector de basura* que se encarga de la limpieza de la memoria y los recursos no utilizados. Limpieza de la memoria del sistema es responsable de la liberación de la memoria dinámica cuando no se utiliza. La creación de muchos objetos, acompañados de numerosas referencias en la memoria es perjudicial porque se llena la memoria y repercute en el rendimiento

global del *recolector de basura* de la máquina virtual.

Suponiendo que hubiera que registrar los números del 1 al 5000 en una variable de tipo **String**. ¿Cómo se puede resolver el problema con el conocimiento que se tiene hasta el momento?

Una de las maneras más simples sería crear una variable para almacenar los números y un bucle de 1 a 5000, en el que cada número es añadido a esa variable. La implementación en Java de la solución sería:

```
String colector = "Números: ";
for (int i = 1; i <= 5000; i++) {
    colector += i;
}
```

El código anterior itera 5000 veces, en cada pasada se añade a `colector` el índice como texto. Salida:

```
Números: 12345678910111213141516 ... 5000
```

Probablemente no se haya notado demasiado el retardo en la ejecución. De hecho, el uso de la concatenación en el bucle ha ralentizado considerablemente la ejecución del programa, llegando quizás a unas décimas de segundo. Puede parecer poco, pero es excesivo, sobre todo si prueba con 50000…

A partir del ejemplo anterior se ve cómo sería un programa de prueba para 50000 números. Se tiene en cuenta el tiempo de ejecución utilizando una *marca de tiempo* obtenida con la clase **Date** antes y después del bucle. Al intentar mostrar el contenido de la variable veremos que no se puede mostrar todo el contenido debido a que la consola tiene un *buffer* de un tamaño limitado y la configuración por defecto no pueden mostrar todo el texto. Si se quiere mostrar todo el valor almacenado, se puede aumentar manualmente la configuración de tamaño del *buffer* de la consola de *Eclipse (Window | Preferencias | Ejecutar / Depurar | Consola)* o guardar el contenido de la variable en un archivo de texto.

```
import java.util.Date;

public class NumerosEncadenados {
	public static void main(String[] args) {
		long marcaTiempo = new Date().getTime();

		String colector = "Números: ";
		for(int i = 1; i <= 50000; i++) {
			colector += i;
```

```
            }
            System.out.println("Tiempo: " + (new Date().getTime() -
                               marcaTiempo) + " ms");
            System.out.println("Total caracteres: " + colector.length());
            System.out.println(colector.substring(0, 50000));
        }
}
```

El problema del excesivo tiempo de ejecución -ya serán varios segundo- está relacionado con el tratamiento de las cadenas en la memoria. Cada iteración tiene que crear un nuevo objeto en el *head* y la actualización de la correspondiente referencia al bloque de memoria. El proceso requiere un tiempo apreciable; sobre todo si se acumula.

En cada paso, suceden varias cosas:

1. Se asigna un bloque de memoria *buffer* para almacenar el resultado de la concatenación. Esta memoria se utiliza sólo de manera temporal hasta que se completa el proceso.
2. Se mueve la antigua cadena al nuevo *buffer* asignado. Si la cadena es larga (digamos 1 MB ó 10 MB), puede ser una operación lenta.
3. Añade otro número como texto en la memoria *buffer*.
4. El bloque de memoria utilizado del *buffer* se convierte en una cadena.
5. Probablemente, el recolector de basura toma el control y limpia la antigua cadena y el *buffer* temporal no utilizado. Este proceso también puede ser lento.

El problema descrito se resuelve de manera mucho más óptima utilizando **`StringBuilder`** o **`StringBuffer`**.

`StringBuilder` y **`StringBuffer`** son clases próximas de **`String`** que proporcionan gran parte de la funcionalidad de la utilización habitual de las cadenas. Representan secuencias de caracteres que se pueden ampliar y modificar. **`StringBuilder`** es la que proporciona mayor rendimiento en el tratamiento de cadenas.

La clase **`StringBuffer`** fue incorporada a partir de la versión 5 de Java. Posee exactamente la misma API que **`StringBuilder`**, salvo que esta última no es *thread-safe* (multihilo con seguridad). En otras palabras, los métodos no se encuentran sincronizados.

Java recomienda siempre que sea posible, utilizar **`StringBuilder`**, dado que es más eficiente que su compañera **`StringBuffer`**. Sólo si se necesita que sea *thread-safe*, se utilizará **`StringBuffer`**.

Las clases **`StringBuilder`** y **`StringBuffer`** se basan en un vector (*array*) de caracteres. Lo importante acerca de ellas es que la información contenida es *mutable*. Los cambios que puedan producir ocurren en el mismo bloque de memoria; se optimizan recursos.

```
//Se crea un StringBuilder (no va al banco de constantes String)
StringBuilder s1 = new StringBuilder("Hola Mundo Java");

s1.append(". Vamos por más");   //Se modifica el mismo objeto (no  se crea un nuevo
                                                    String)
```

Si se comparan los tiempos del ejemplo anterior en el que se concatenan el número en una cadena **`String`** con un bucle con la solución basada en **`StringBuilder`**, se puede comprobar que se tarda del orden de muchos cientos de veces menos:

```
import java.util.Date;

public class NumerosEcadenadosSB {
	public static void main(String[] args) {
		long marcaTiempo = new Date().getTime();

		StringBuilder colector = new StringBuilder("Números: ");
		for(int i = 1; i <= 50000; i++) {
			colector.append(i);
		}
		System.out.println("Tiempo: " + (new Date().getTime() - marcaTiempo) +
	" ms");
		System.out.println("Total caracteres: " + colector.length());
		System.out.println(colector.substring(0, 50000));
	}
}
```

append()

El método `append()` de **`StringBuilder`** y **`StringBuffer`** es llamado automáticamente a menudo a través del operador **+**. Tiene versiones sobrecargadas para todos los tipos primitivos. Se llama automáticamente a **`String`**`.valueOf()` para cada parámetro y el resultado se añade al **`StringBuilder`** / **`StringBuffer`** actual. Cada versión de `append()` devuelve su propio *buffer*.

Resumen de métodos de StringBuilder y StringBuffer

Tipo retorno	Método	Sobrecarga	Descripción	Ejemplo
StringBuffer/ StringBuilder	append(String s)	SI	Concatena la cadena s a la cadena dentro del objeto (no es necesario atrapar la referencia para que	`StringBuffer s = new StringBuffer("Quiero ser"); s.append(" un SCJP!"); //Quiero ser un SCJP!`

			el cambio se realice).	
StringBuffer/ StringBuilder	`setCharAt(int indice, char c)`	SI	Cambia un carácter de la cadena dada la posición indicada por índice dentro de la cadena del objeto.	`StringBuffer s = new StringBuffer("Quiero SCJP"); s.charAt(8, 'H'); //Quiero HCJP`
StringBuffer/ StringBuilder	`delete(int inicio, int fin)`	NO	Elimina la subcadena dentro de la cadena del objeto desde la posición inicio a la posición fin.	`StringBuffer s = new ngBuffer("Quiero ser CJP!"); s.delete(10, //Quiero ser SCJP!`
StringBuffer/ StringBuilder	`insert(int indice, String s)`	SI	Inserta una nueva cadena (s) a partir de la posición indicada por índice dentro de la cadena del objeto.	`StringBuffer s = new StringBuffer("Quiero SCJP"); s.insert(7, "ser un "); //Quiero ser un SCJP`
StringBuffer/ StringBuilder	`reverse()`	NO	Invierte la cadena.	`StringBuffer s = new StringBuffer("PJCS"); s.revert(); //SCJP`
String	`toString()`	NO	Devuelve un **`String`** con el valor del objeto.	`StringBuffer s = new StringBuffer("SCJP"); String s1 = s.toString(); //SCJP`

Búsqueda, parseo y validación con expresiones regulares

Para la búsqueda en grandes cadenas de texto, lo más común, y utilizado, son las expresiones regulares; representan un lenguaje específico para el tratamiento de texto.

Una expresión regular define un patrón de búsqueda dentro de una cadena de texto. Las expresiones regulares se pueden utilizar para comprobar si una cadena o subcadena dada coincide con el patrón o esquema especificado. **En la cadena objeto de la búsqueda pueden aparecer ninguna o más concordancias con el patrón.**

Algunas situaciones en las que es adecuado el uso de expresiones regulares son:

- ➢ Para comprobar que una fecha cumple el patrón dd/mm/aaaa.
- ➢ Para comprobar que un NIF está formado por 8 cifras y una letra.
- ➢ Para comprobar que una dirección de correo electrónico es una dirección válida.

- ➢ Para comprobar que una contraseña cumple unas determinadas condiciones.
- ➢ Para comprobar que una URL es válida.
- ➢ Para comprobar cuántas veces se repite dentro de la cadena una secuencia de caracteres determinada.

Las expresiones regulares son como pequeños programas que **permiten identificar una sección de texto que cumpla con ciertas características o patrón; este patrón se busca en el texto de izquierda a derecha**. Cuando se determina que un carácter cumple con el patrón este carácter ya no vuelve a intervenir en la comprobación.

Ejemplo:

La expresión regular: `"010"`
Se encuentra dentro del `texto: "0101010"` dos veces; sin solapar concordancias.

Los lenguajes que tienen soporte para expresiones regulares permite buscar una determinada expresión. Lo que se hace es pasar al *procesador regex* una cadena y una expresión regular.

Creación de expresiones regulares

Hay que tener en cuenta que en Java se deben usar una doble barra invertida **** en cualquier punto donde se requiera una como parte de la expresión. Por ejemplo para utilizar **\w** tendremos que escribir **\\w**. Si se quiere indicar que la barra invertida es un carácter de la expresión regular habrá que escribir ****.

Expresión	Descripción	Ejemplo
asd	Cuando se especifican caracteres normales, sin un significado especial en la sintaxis regex, se interpretan como una cadena simple que se busca dentro del texto analizado (diferencia mayúsculas de minúsculas).	Cadena: "aa.ssdd.**asd**dd.**asd**" Expresión: **asd** Coincidencias: 8, 14
.	Un punto en una expresión representa un carácter cualquiera.	Cadena: "abc del **mare**o" Expresión: **mar.** Coincidencia: 8
\.	Cuando se quiere especificar el punto como carácter del texto, se debe indicar con una barra invertida. En Java la barra debe ser doble: \\.	Cadena: "aa.ssdd.asddd.asd" Expresión: **\.** Coincidencias: 2, 7, 13
^expresión	El símbolo **^** indica el principio del String. En este caso el String debe empezar por los caracteres indicados por la expresión .	Cadena: "**aa**.ssdd.aaddd.asd" Expresión: **^aa**

		Coincidencia: 0
expresión$	El símbolo **$** indica el final del String. En este caso el String debe terminar con los caracteres indicados en la expresión.	Cadena: "aa.ssdd.asddd.a**sd**" Expresión: **sd$** Coincidencias: 15
\d **\D** **\s** **\S** **\w** **\W** **\b**	Estos metacaracteres buscan: (En Java la barra debe ser doble: ****) **\d** Cualquier número (**0** a **9**). Equivale a: **[0-9]** **\D** Cualquier carácter no numérico. Equivale a: **[^0-9]** **\s** Espacios en blanco y saltos de línea. Equivale a: **[\t\n\r\f\x0b]** **\S** No espacios en blanco o saltos de línea. Equivale a: **[^\s]** **\w** Cualquier carácter alfanumérico (letras, números, o _) . Equivale a: **[a-zA-Z0-9_]** **\W** Cualquier carácter no alfanumérico . Equivale a: **[^\w]** **\b** Límite de una palabra. Indica en qué posición empieza o acaba una palabra separada por espacios en blanco o caracteres especiales.	Cadena: " **0123** aadd_ d**6**df" Expresión: **\d** Coincidencias: 1, 2, 3, 4, 15 Cadena: " 0123 **aadd_** d6**df**" Expresión: **\D** Coincidencias: 0, 5, 6,7, 8, 9, 10, 11, 12, 13, 14, 16, 17 Cadena: " 0123 aadd_ d6df" Expresión: **\s** Coincidencias: 0, 5, 11, 12, 13 Cadena: " 0123 aadd_ d6df" Expresión: **\W** Coincidencias: 0, 5, 11, 12, 13 Cadena: " 0123 aadd_ d6df" Expresión: **\b** Coincidencias: 1, 5, 6, 11, 14, 16
[abc], **[a-f],** **[a-fA-F]**	Los corchetes representan la definición de un conjunto de caracteres posibles que se buscan para que haya coincidencia.	Cadena: "J**a**v**a**Worl**d**" Expresión: **[0-9a-f]** Coincidencias: 1, 3, 8

	Dentro de los **[]** se especifica un juego y/o rango de caracteres válidos posibles. Aquellos comprendidos entre **–** son los rangos. Ejemplo: Busca cualquier dígito **[0123456789]**, o lo que es lo mismo **[0-9]**.	
[^asd0-9]	El símbolo ^ dentro de los corchetes indica negación. La expresión **[^**…**]** actúa al contrario que **[**…**]**. Ésta se comporta buscando cualquier carácter que no se encuentre dentro del rango especificado.	Cadena: "0123**45**6789" Expresión: **[^0-36-9]** Coincidencias: 4, 5
()	Indican un grupo de caracteres que pueden aparecer. Dentro del grupo puede haber opcionalidad indicada con corchetes.	Cadena: "Jav**aW**orld" Expresión: **(a[Ww])** Coincidencia: 3 Cadena: "**pepe@gmail.com**" Expresión: **([^@])+** Coincidencias: 0, 5
A|B	El carácter **|** es un OR. A ó B	
aB	Concatenación a seguida de B	
+, *****, **?** **{N}** **{N,M}**	**Comodines de cantidad y cuantificadores.** Permiten indicar una cantidad de un grupo o carácter a buscar. **+** indica al menos 1 o más. Equivale a: {1,} ***** indica 0 o más. Equivale a: {0,} **?** indica 0 o 1 ocurrencia. Equivale a: {0,1} **{N}** Indica que lo que va justo antes de las llaves se repite N veces. **{N,M}** Indica que lo que va justo antes de las llaves se repite mínimo N veces y máximo M veces. También podemos poner {N,} indicando que se repite un mínimo de N veces sin límite máximo. Cuando se busca un carácter simple con ?, hay que tener en cuenta que después del último carácter útil hay un final de cadena que entra dentro de la selección por el ? como 0 ocurrencias. En las combinaciones ***?** , **+?** y **{N,M}?**, El ? actúa forzando la combinación más corta que concuerda con el patrón.	Cadena: "01 a**2**a **223**" Expresión: **\d+** Coincidencias: 4, 7 Cadena: "**b**a**b**" Expresión: **b?** Coincidencias: 0, 2 Cadena: "**b**a**b**" Expresión: **[b?]** Coincidencias: 0, 2

Para usar expresiones regulares en Java se usa el package `java.util.regex` que contiene las clases `Pattern` y `Matcher` y la excepción `PatternSyntaxException`.

- Un objeto de la clase `Pattern` representa la expresión regular. La clase `Pattern` contiene el método `compile(String regex)` que recibe como parámetro el `String` con la expresión regular y devuelve un objeto de la clase `Pattern`.
- La clase `Matcher` compara el `String` y la expresión regular. Contienen el método `matches(CharSequence input)` que recibe como parámetro el `String` a validar y devuelve **`true`** si coincide con el patrón. El método `find()` indica si el `String` contiene el patrón.

Ejemplos de expresiones regulares en Java

➔ Comprueba si la cadena contiene exactamente el patrón "asd"

Cadena: "aa.ssdd.**asd**dd.**asd**"
Expresión: **asd**
Coincidencias: 8, 14

➔ Comprueba si la cadena empieza exactamente con el patrón "^aa"

Cadena: "**aa**.ssdd.aaddd.asd"
Expresión: **^aa**
Coincidencias: 0

➔ Comprueba si la cadena termina exactamente con el patrón "sd$"

Cadena: "aa.ssdd.asddd.a**sd**"
Expresión: **sd$**
Coincidencias: 15

➔ Comprueba si la cadena contiene puntos con el patrón "\."

Cadena: "aa.ssdd.asddd.asd"
Expresión: **\\.**
Coincidencias: 2, 7, 13

➔ Comprueba si la cadena contiene los tres caracteres indicados con el patrón "mar", seguido de cualquier carácter.

Cadena: "abc del **mare**o"
Expresión: **mar.**
Coincidencias: 8

➔ Comprueba si la cadena contiene caracteres numéricos con el patrón "\d"

Cadena: " **0123** aadd_ d**6**?f"
Expresión: **\\d**
Coincidencias: 0, 1, 2, 3, 15

➔ Comprueba si la cadena tiene espacios en blanco, saltos y tabuladores con el patrón "\s"

Cadena: "**0**123 **a**add_ **d**1?**f**"

Expresión: **\\s**
Coincidencias: 0, 5, 11, 12, 13

➔ Comprueba si la cadena tiene caracteres no alfabéticos "\W". Incluye todos los caracteres especiales.
Cadena: **"** 0123 aadd_ d1**?**f**"**
Expresión: **\\W**
Coincidencias: 0, 5, 11, 12, 13, 16

➔ Comprueba si la cadena tiene separadores de palabras "\b"
Cadena: **"** **0**123 **a**add_ **d**1**?f"**
Expresión: **\\b**
Coincidencias: 1, 5, 6, 11, 14, 16, 17, 18

➔ Comprueba si la cadena contiene [0-9a-f]
Cadena: "J**a**v**a**Worl**d"**
Expresión: **[0-9a-f]**
Coincidencias: 1, 3, 8

➔ Comprueba si la cadena contiene el grupo a seguida de una w o W
Cadena: "Jav**aW**orld"
Expresión: **(a[Ww])**
Coincidencias: 3

➔ Comprueba si la cadena contiene exactamente el patrón "asd"
Cadena: "aa.ssdd.**asd**dd.**asd"**
Expresión: **(asd)**
Coincidencias: 8, 14

➔ Comprueba si la cadena contiene uno o varios grupos formados por caracteres que no sean @
Cadena: "pepe@gmail.com"
Expresión: **([^@])+**
Coincidencias: 0, 5

➔ Comprueba dónde aparece -o no- "b".
Cadena: **"bab"**
Expresión: **b?**
Coincidencias: 0, 1, 2

➔ Comprueba dónde aparece "b".
cadena: **"b**a**b"**
Expresión: **[b?]**
Coincidencias: 0, 2

➔ Comprueba si la cadena contiene al menos un grupo de caracteres numéricos.
Cadena: **"0**1 a**2**a **2**23**"**
Expresión: **\\d+**
Coincidencias: 0, 4, 7

➔ Comprueba si la cadena contiene el patrón “abc”, opcionalmente seguido de ninguno o varios caracteres.
Cadena: **"abcd ggg"**
Expresión: **abc.***
Coincidencia: 0 (cadena completa)

➔ Comprueba si la cadena contiene “abc”, opcionalmente precedido o seguido de ninguno o varios caracteres.
Cadena: **"sdfgabcd ggg"**
Expresión: **.*abc.***
Coincidencia: 0 (cadena completa)

➔ Comprueba si la cadena empieza por “abc”, opcionalmente seguido de ninguno o varios caracteres.
Cadena: **"abc del mar"**
Expresión: **^abc.***
Coincidencia: 0 (cadena completa)

➔ Comprueba si la cadena empieza por “abc” ó “Abc”, opcionalmente seguido de ninguno o varios caracteres.
Cadena: **"abc dfdgfdg asdrabcty"**
Expresión: **^[aA]bc.***
Coincidencia: 0 (cadena completa)

➔ Comprueba si en la cadena hay patrones formados por un mínimo de 5 letras mayúsculas o minúsculas y un máximo de 10.
Cadena: "abc **dfdgfdg asdrabcty"**
Expresión: **[a-zA-Z]{5,10}**
Coincidencias: 4, 12

➔ Comprueba si la cadena no empieza por un dígito, opcionalmente seguido de ninguno o varios caracteres.
Cadena: **"abc dfdgfdg asdrabcty"**
Expresión: **^[^\\d].***
Expresión: **^[^\\d].{0,}**
Coincidencia: 0 (cadena completa)

➔ Comprueba si la cadena no acaba con un dígito, opcionalmente precedido de ninguno o varios caracteres.

Cadena: **"abc dfdgfdg asdrabcty"**
Expresión: **.*[^\\d]$**
Coincidencia: 0 (cadena completa)

➔ Comprueba si la cadena contienen los caracteres a ó b.

Cadena: **"a**bc dfdgfdg **a**sdrabcty**"**
Expresión: (a|b)+
Coincidencias: 0, 12, 16

➔ Comprueba si la cadena contiene un 1 y ese 1 no está seguido por un 2, opcionalmente va precedido o seguido de ninguno o varios caracteres.

Cadena: **"abc dfdg13fdg asdrabcty"**
Expresión: **.*1[^2].***
Expresión: **.*1(?!2).***
Coincidencia: 0 (válida)

➔ Comprueba si un correo electrónico es válido. Como referencia de los caracteres admitidos en una dirección de correo electrónico Ver: https://www.jochentopf.com/email/chars.html

Cadena: **"pepe.al@gmail.com"**
Expresión:
^[\\w-\\+]+(\\.[\\w-\\+]+)*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$
Coincidencia: 0 (válida)

Cada parte de la expresión regular es la siguiente:

^[\\w-\\+]+	➢ Primer carácter de una dirección de email, indicado con **^** ➢ El signo **+** indica que debe aparecer uno o más de los caracteres indicados entre corchetes. ➢ **\\w** indica los caracteres de la A a la Z, tanto mayúsculas como minúsculas, dígitos del 0 al 9 y el carácter _ ➢ Se incluye el carácter **-** como carácter válido. ➢ Se incluye el carácter **+** como carácter válido. ➢ Alternativamente, en lugar de \\w, se podría haber escrito el rango de caracteres; con lo que esta expresión quedaría: [A-Za-z0-9-_\\+]+
(\\.[\\w-\\+]+)*	➢ A partir del primero o primeros caracteres de una dirección de email. ➢ Los paréntesis indican un grupo formado por:

	○ El carácter **.** ○ El **+** indica que al menos debe aparecer un carácter de los incluidos entre corchetes: ■ **\\w** indica caracteres de la A a la Z, tanto mayúsculas como minúsculas, dígitos del **0** al 9 y el carácter _ ■ Se incluye el carácter **–** como carácter válido. ■ Se incluye el carácter **+** como carácter válido. ➢ El * indica que el grupo es opcional, puede no aparecer o hacelo varias veces.
@	➢ El carácter **@** siempre debe aparecer.
[A-Za-z0-9]+	➢ Después de **@** ➢ El **+** indica que debe aparecer al menos uno o más de los caracteres indicados entre corchetes: ○ Caracteres alfabéticos, mayúsculas o minúsculas y dígitos del **0** al **9.**
(\\.[A-Za-z0-9]+)*	➢ Seguido, los paréntesis indican un grupo formado por: ○ El carácter **.** ○ El **+** indica que debe aparecer al menos uno o más de los caracteres indicados entre corchetes: ■ Caracteres alfabéticos, mayúsculas o minúsculas y dígitos del **0** al **9.** ➢ El * indica que el grupo es opcional, puede no aparecer o hacelo varias veces.
(\\.[A-Za-z]{2,})$	➢ Al final de una dirección de correo electrónico, indicado por el **$**, debe aparecer un grupo -indicado por los paréntesis- formado por: ○ El carácter **.** ○ El **{2,}** indica que debe aparecer al menos dos o más de los caracteres indicados entre corchetes: ■ Caracteres alfabéticos, mayúsculas o minúsculas**.**

➔ Comprueba si una contraseña tiene una longitud mínima de 5 caracteres alfanuméricos, incluida la ñ, y los caracteres especiales: **$*-+&!?**

Cadena: **"Prueba9x$"**
Expresión: **([\\wñÑ$*-+&!?]){5,}**
Coincidencia: 0 (válida)

➔ Comprueba si una contraseña es robusta asegurando una longitud mínima de 8 caracteres entre los cuales se encuentran al menos:

- Una letra mayúscula.
- Una letra minúscula.
- Un dígito numérico.
- Un carácter especial de los especificados: **`$*-+&!?%`**

Cadena: **`"PrueBa89x$"`**
Expresión:
`(?=.*[A-ZÑ])(?=.*[a-zñ])(?=.*\\d)(?=.+[$*-+&!?%]).{8,16}`
Coincidencia: 0 (válida)

Ejemplo: Uso del procesador de *regex* en Java

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class PruebasRegex {

	static public void main(String[] args) {
		String texto = "012abc";	//Ejemplo
		String regex = "\\d";		//Ejemplo

		Pattern patron = Pattern.compile(regex);
		Matcher concordancias = patron.matcher(texto);

		System.out.print("Posiciones:\t");
		for (int x = 0; x < texto.length(); x++) {
			if (x < 10) {
				System.out.print(x + "  ");
			}
			else {
				System.out.print(x + " ");
			}
		}

		System.out.print("\nTexto:\t\t");
		for (char c : texto.toCharArray()) {
			System.out.print(c + "  ");
		}

		System.out.println("\nRegex: " + concordancias.pattern());

		System.out.println("\nConcordancias: ");
		while (concordancias.find()) {
			System.out.println("[" + concordancias.start() + "," +
			concordancias.end() + "]: " + concordancias.group());
```

```
            }
        }
}
```

Salida:

```
Posiciones:   0 1 2 3 4 5
Texto:        0 1 2 a b c
Regex: \d
Coincidencias:
[0,1]: 0
[1,2]: 1
[2,3]: 2
```

Grupos y referencias inversas

Se pueden agrupar partes del patrón de una expresión regular con paréntesis. Los paréntesis normalmente permiten asignar un operador de repetición para un grupo completo. Estos grupos también crean una referencia inversa a la parte de la expresión regular correspondiente.

Las referencias inversas a los grupos permiten acceder a las distintas partes del texto original concordantes con los grupos para cualquier fin. El carácter `$` permite hacer referencia a un grupo. Por Ejemplo `$1` es el primer grupo, `$2` el segundo, etc.

Ejemplo: *Se quieren quitar todos los espacios en blanco que anteceden a un punto o una coma en un texto*. Esto implicaría que el punto o la coma es parte del patrón. Aún así, debe incluirse en el resultado.

```
// Elimina los espacios en blanco entre una letra seguida de un punto o una coma
String regex = "(\\w)(\\s+)([\\.,])";
System.out.println("Hola  , Esto está regular .".replaceAll(regex, "$1$3"));
```

En este ejemplo se extrae el texto entre las etiqueta.

```
// Extrae el texto entre las etiquetas
regex = "(?i)(<titulo.*?>)(.+?)(</titulo>)";
String actualizada = "<titulo>EJEMPLO_TEST</titulo>".replaceAll(regex, "$2");
```

Búsqueda y separación en tokens

***Tokenizing* es el proceso por el cual se separa un bloque de texto en varios *tokens*, los cuales se encuentran separados por uno o varios caracteres especiales, denominado *parser* (separador).**

Ejemplo, la secuencia de texto `"aa2d3,222,333",` suponiendo que el carácter de *parser* es la coma (,) daría la salida:

```
aa2d3
222
333
```

Obtener tokens con la clase Scanner

A diferencia de **`Pattern`**, la clase **`Scanner`** no posee funcionalidad para eliminar o reemplazar; es posible buscar una expresión regular y obtener la cantidad de veces que esta aparece.

Cuando es necesario realizar una búsqueda de *tokens* avanzada, la clase `java.util.`**`Scanner`** es una buena opción. Algunas de sus ventajas son:

- Un objeto **`Scanner`** puede ser construido a partir de: `String`, `Stream` o `File`.
- La búsqueda de *tokens* se realiza dentro de un bucle, pudiendo salir de este cuando se desee.
- Los *tokens* pueden ser convertidos a un tipo primitivo inmediatamente y almacenados en cualquier sitio.

Algunos de los métodos de la clase **`Scanner`** son:

- `nextXxx()` Xxx puede ser reemplazado por el nombre de un tipo primitivo de datos. Devuelve el *token* formateado con el tipo de dato especificado.
- `next()` Devuelve el *token* como `String`.
- `hasNextXxx()` Xxx puede ser reemplazado por el nombre de un tipo primitivo de datos. Verifica si el siguiente *token* es del tipo de datos especificado.
- `hasNext()` Indica si hay otro *token* más para procesar.
- `useDelimiter(String regex)` Utiliza una expresión regular como argumento en la llamada para utilizarla como *parser o separador*.

```java
import java.util.Scanner;
public class PruebasScanner {
	static public void main(String[] args) {
		String texto = "Estamos bebiendo vino ALBARIÑO..."
				+ "No tenemos más   qué decir.";
		Scanner sc = new Scanner(texto);
		String token;
		//Separa las palabras del texto...
		do {
			token = sc.findInLine("[\\wñÑáéíóú]+");
			System.out.println("Coincidencia: " + token);
		} while (token != null);
	}
```

```
} //class
```

Salida:

```
Coincidencia: Estamos
Coincidencia: bebiendo
Coincidencia: vino
Coincidencia: ALBARIÑO
Coincidencia: No
Coincidencia: tenemos
Coincidencia: más
Coincidencia: qué
Coincidencia: decir
Coincidencia: null
```

Obtener tokens con el método String.split(regex)

El método `String.split(String regex)` recibe como parámetro una *expresión regular*. Lo que hará `split` es generar un *array* con los *tokens* que se encuentran delimitados por el *parser* especificado.
El método `Split()` es la alternativa a usar la clase `StringTokenizer` para separar cadenas. Este método trocea el `String` en subcadenas según la expresión regular que recibe. La expresión regular no se incluirá en el array resultante.

Ejemplo:

```
public class PruebasSplit1 {
	static public void main(String[] args) {
		//Corta utilizando .
		String texto = "hola.Java.World";
		String[] cadenas = texto.split("\\.");

		for (String str : cadenas) {
			System.out.println(str);
		}
	}
}
```

Salida:

```
hola
Java
World
```

Ejemplo:

```
public class PruebasSplit2 {
	static public void main(String[] args) {
		//Corta utilizando cualquiera de los caracteres - : . _
		String texto = "blanco-rojo:amarillo.verde_azul";
		String[] cadenas = texto.split("[-:\\._]");

		for (String str : cadenas) {
			System.out.println(str);
		}
	}
}
```

Salida:

```
blanco
rojo
amarillo
verde
azul
```

Ejemplo:

```
public class PruebasSplit3 {
	static public void main(String[] args) {
		//corta utilizando e seguido de s o m
		//también corta utilizando pl
		String texto = "esto es un ejemplo de cómo funciona split";
		String[] cadenas = str.split("(e[sm])|(pl)");

		for (String str : cadenas) {
			System.out.println(str);
		}
	}
}
```

Salida:

```
to
un ej
o de cómo funciona s
it
```

El inconveniente de este enfoque es que se aplica a toda la cadena. Para textos grandes no tiene mucho rendimiento y menos, si lo que se necesita está al principio del texto.

Validación de formato con String.matches(regex)

Se puede comprobar si una cadena de caracteres cumple con un patrón de formato usando el método `matches()` de la clase `String`. Este método recibe como parámetro la expresión regular.

Ejemplo:

```
public class PruebasValidacion {
	static public void main(String[] args) {
		//Comprueba si el formato lleva un 1 y no lleva consecutivo un 2,
		//opcionalmente puede ir precedido o seguido de ninguno o varios
		caracteres.
		String texto = "abc dfdg132fdg asdrabcty";
		if (texto.matches(".*1[^2].*")) {  //Equivalente: .*1(?!2).*
		   System.out.println("Cumple formato");
		}
		else {
		   System.out.println("No cumple formato");
		}
	}
}
```

Salida:

```
Cumple formato
```

Sustitución con String.replaceAll(regex, reemplazo)

Reemplaza con el texto de reemplazo las coincidencias obtenidas con un patrón regex.
Este método recibe como parámetro la expresión regular y el texto de reemplazo.

Ejemplo:

```
public class PruebasReplace {
	static public void main(String[] args) {
		//Reemplaza cualquier letra a y sus dos siguiente caracteres por AAA
		String texto = "abc dfdg13fdg asdrabcty";
		System.out.println(texto.replaceAll("a..", "AAA"));
	}
}
```

Salida:

```
AAA dfdg13fdg AAArAAAty
```

Formateo con printf() y format()

La clase `java.io.`**`PrintStream`** proporciona varias opciones para presentar la salida de datos de manera formateada. Hay que aclarar que `printf()` y `format()` son métodos exáctamente iguales (corre el rumor de que *Sun* introdujo `printf()` para agradar a los programadores de C++), de manera que se verá todo con `format()`, siendo equivalente la sintaxis para `printf()`. La firma del método es:

```
format(String formato, Object... argumentos)
```

El `String formato` es una expresión que admite caracteres especiales para el formateo de los datos y tiene la siguiente estructura:

`%[INDICE_ARGUMENTO$][FLAGS][ANCHO][.PRESICION]`**`FORMATO`**

No lleva ningún espacio, y todos los argumentos dentro de `[]` son opcionales.

Resumen de los posibles valores para la expresión de formato

Parámetro	Valor	Descripción	Ejemplo
INDICE_ARGUMENTO	int	Indica a qué posición hace referencia de los argumentos especificados. Si no se especifica, se administran por orden de primero (izquierda) a último (derecha). El primer argumento corresponde al índice 1.	`format("%2$d - %1$d", 12, 15);` **Salida:** `15 - 12`
**FLAGS	-	*Alinea a la izquierda (**requiere especificación de ANCHO**).	`format("int: %-15d \n", 123);` **Salida:** `int: 123`

	+	***Requiere que FORMATO sea un número (d | f)**, incluye el signo del número (+ o -).	`format("int: %+d \n", 123);` **Salida:** `int: +123`

	0	***Requiere que FORMATO sea un número (d | f)**, agrega ceros a la izquierda (**requiere especificación de ANCHO**). En los `float` la coma ocupa un espacio.	`format("int: %05d \n", 123);`
			Salida: `int: 00123`
			
			`format("float: %012f \n", 1.23f);`
			Salida: `float: 00001,230000`
			
			`format("float: %05.2f \n", 1.23f);`
			Salida: `01,23`

	,	***Requiere que FORMATO sea un número (d | f)**, indica que se utilice el separador de decimales configurado para el país.	`format("float: %,f \n", 1.23f);` **Salida:** `float: 1,230000`
	(	***Requiere que FORMATO sea un número (d | f)**, encierra los valores negativos entre paréntesis.	`format("float: %(f \n", -1.23f);` **Salida:** `float: (1,230000)`
ANCHO	int	Indica la cantidad mínima de valores a escribir. (Por defecto rellena con espacios, salvo que se especifique FLAGS **0**, y con alineación a la derecha, salvo que se especifique FLAGS **-**).	`format("int: %5d \n", 123);` **Salida:** `int:  123`
PRECISION	int	***Requiere que FORMATO sea un número de punto flotante (f)**, indica la cantidad de decimales a mostrar.	`format("float: %.2f \n", 1.23f);` **Salida:** `float: 1,23`
FORMATO	b	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **`boolean`**.	`format("boolean: %b", true);` **Salida:** `boolean: true`
	c	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **`char`**.	`format("char: %c", 'Z');` **Salida:** `char: Z`

	d	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **`int`**.	`format("int: %d", 123);` **Salida:** `int: 123`
	f	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **`float`**.	`format("float: %f", 1.23f);` **Salida:** `float: 1,230000`
	s	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo `String`.	`format("String: %s", "JavaWorld");` **Salida:** `String: JavaWorld`

*Si no se cumple lo resaltado en negrita al aplicar el parámetro, se generará un error en tiempo de ejecución, lanzando una excepción dependiendo del parámetro faltante o erróneo, o error de conversión en caso del parámetro FORMATO.
**Los flags son combinables, pudiendo especificar más de uno al mismo tiempo.

Ejercicios

1.
   - ➢ Copia y prueba todos los ejemplos sobre formateo de salida en Java con los métodos format() y printf() que se proporcionan
   - ➢ Escribe una variante de cada uno de los ejemplos.
   - ➢ Documenta con comentarios aclaratorios adicionales.

2.
   - ➢ Escribe un método que se llame `invertirTexto()` que reciba una cadena de caracteres cualquiera y la vuelve invertida. Por ejemplo:
     - o `"Introducción" -> "nóiccudortnI"`.
   - ➢ Prueba el método pedido desde `main()`.
   - ➢ Se recomienda utilizar internamente un **`StringBuilder`** y un bucle `do-while`.
   - ➢ Documenta el código fuente con comentarios aclaratorios adicionales.

3
   - ➢ Escribe un método que se llame `verificarParentesis()` que recibe una cadenas de texto que contiene una expresión aritmética en la que hay que comprobar que los paréntesis están bien emparejados. Devuelve verdadero o falso. Por ejemplo, si se proporciona:

     Expresión: `((a + b) / 5-d)` Daría: `true`

     Expresión: `)(a + b))` Daría: `false`
   - ➢ Prueba el método pedido desde `main()`.
   - ➢ Utiliza un contador para los paréntesis: Cuando se abre un paréntesis incrementa el contador, en cuando se abre se decrementa en `1`. Al final el contador debe valer `0`; en cualquier otro caso la expresión es incorrecta.
   - ➢ Documenta el código fuente con comentarios aclaratorios adicionales.

4.
   - ➢ Escribe un método que se llame `vecesSubCadena()` que recibe dos cadenas y devuelve el número de veces que la segunda cadena está contenida en el texto de la primera. Por ejemplo, si se busca la subcadena `"en"` en el texto:

     "Estamos viviendo en un submarino amarillo. No tenemos nada que hacer. En el interior del submarino se está muy apretado. Así que estamos leyendo todo

el día. Vamos a salir en 5 días.

Daría: `5`

- ➢ Prueba el método pedido desde `main()`.
- ➢ Se recomienda probar el texto de ejemplo y utilizar un bucle de búsqueda que utilice la versión adecuada de **`indexOf()`** con un índice.
- ➢ Documenta el código fuente con comentarios aclaratorios adicionales.

5.

- ➢ Escribe un método que se llame `vecesSubCadena2()` que recibe dos cadenas y devuelve el número de veces que la segunda cadena está contenida en el texto de la primera sin distinguir mayúsculas . Por ejemplo, si se busca la subcadena `"en"` en el texto:

  "Estamos viviendo en un submarino amarillo. No tenemos nada qué hacer. En el interior del submarino se está muy apretado. Así que estamos leyendo todo el día. Vamos a salir en 5 días.

  Daría: `6`

- ➢ Prueba el método pedido desde `main()`.
- ➢ Se recomienda probar el texto de ejemplo y utilizar un bucle de búsqueda que utilice la versión adecuada de **`indexOf()`** con un índice.
- ➢ Documenta el código fuente con comentarios aclaratorios adicionales.

6.

- ➢ Escribe un método que se llame `mayusSubCadena()` que reciba un texto etiquetado y devuelve otra cadena en la que se han cambiado a mayúsculas todos los caracteres en el texto entre las etiquetas **`<mayus>`** y **`</mayus>`**. Las etiquetas no se pueden anidar y deben ser limpiadas del texto resultante:

  "Estamos viviendo en un <mayus>submarino amarillo</mayus>. No tenemos <mayus>nada</mayus> qué hacer.

  Daría:

  "Estamos viviendo en un SUBMARINO AMARILLO. No tenemos NADA qué hacer"

- ➢ Prueba el método pedido desde `main()`.
- ➢ Se recomienda utilizar expresiones regulares o **`IndexOf()`** para abrir y cerrar la etiqueta. Una vez calculado el índice de inicio y final del texto afectado por una etiqueta se extrae, se pasa a mayúscula y se reemplaza toda la subcadena **etiqueta**

de apertura + texto + etiqueta de cierre.

➢ Documenta el código fuente con comentarios aclaratorios adicionales.

7.

➢ Escribe un método que se llame `padRight()` que reciba una cadena de caracteres, un carácter y un número. Devuelve una cadena formateada con el texto recibido completado por la derecha con el carácter proporcionado hasta la longitud indicada con el número. Por ejemplo:

```
padRight("Introducción", '*', 20);
```

produce: `"Introducción********"`

➢ Prueba el método pedido desde `main()`.
➢ Se recomienda utilizar `StringBuilder`.
➢ Documenta el código fuente con comentarios aclaratorios adicionales.

8.

➢ Escribe un método que se llame `reemplazaPalabras()` que reciba una cadena con un texto y otra con una serie de términos separados por comas que deben ser sustituidos por asteriscos. Devuelve el texto con el correspondiente cambio. Por ejemplo para el texto:

```
"Oracle ha anunciado hoy su nueva generación de
compilador Java. Utiliza analizador avanzado y
optimizador especial para la JVM de Oracle"
```

Términos: `"JVM,Java,Oracle"`

```
"****** ha anunciado hoy su nueva generación de
compilador ****. Utiliza analizador avanzado y
optimizador especial para la *** de ******"
```

➢ Prueba el método pedido desde `main()`.
➢ Se recomienda utilizar `StringBuilder` y el método `split()` de `String` para separar cada término a rastrear. Al encontrar una palabra a sustituir por asteriscos se deben respetar las longitudes de cada término .
➢ Documenta el código fuente con comentarios aclaratorios adicionales.

9.

➢ Copia y prueba todos los ejemplos sobre expresiones regulares en Java que se proporcionan en el **Manual de Java**, en el tema: ***Cadenas de caracteres.***
➢ Escribe una variante de cada uno de los ejemplos.

➢ Documenta con comentarios aclaratorios adicionales.

10.

➢ Escribe una nueva versión de la clase **PruebasRegex**, dada como ejemplo de uso del procesador de expresiones regulares en Java, para que utilice los dos argumentos de la línea de comandos del sistema para introducir la expresión regular y el texto a procesar.
➢ Los argumentos de la línea de comandos se recibe, al ejecutar un programa, como un array de `String` que se especifica en todos los programas Java en el método: `static public void main(String[] args)`
➢ Documenta el código fuente con comentarios aclaratorios adicionales.

11.

➢ Escribe un método que se llame `validarFormatoContraseña()` que reciba una cadena de caracteres correspondiente a una cotraseña y devuelve **`true`** o **`false`** según se cumplan las siguientes condiciones:
  1. Longitud: entre `5` y `25`
  2. Caracteres admitidos:
     - ❏ Alfanuméricos incluida la ñ
     - ❏ Caracteres especiales: `$, *, -, +, !, ?`

➢ Documenta el código fuente con comentarios aclaratorios adicionales.

12.

➢ Escribe un método que se llame `separaURL()` que reciba una cadena de caracteres de una URL y devuelve un array de tres `String` conteniendo el *protocolo*, el *servidor* y el *recurso* de la URL recibida. Por ejemplo, cuando se proporciona `http://www.devbg.org/forum/index.php` el resultado es:

```
resultado[0]: "http"              (Protocolo)
resultado[1]: "www.devbg.org"     (Servidor)
resultado[2]: "/forum/index.php"  (Recurso)
```

➢ Prueba el método pedido desde `main()`.
➢ Utiliza expresiones regulares o busca los respectivos separadores: dos barras diagonales para poner fin al protocolo y una barra diagonal como separador entre el servidor y el recurso.
➢ Documenta el código fuente con comentarios aclaratorios adicionales.

13.

- Escribe un método que se llame `validarContraseñaCompleja()` que reciba una cadena de caracteres correspondiente a una contraseña en la que se quiere asegurar un nivel de complejidad. Devuelve **`true`** o **`false`** según se cumplan las siguientes condiciones:
  1. Longitud mínima: `8`.
  2. Longitud máxima ilimitada.
  3. Al menos un carácter alfabético en mayúscula.
  4. Al menos un carácter alfabético en minúscula.
  5. Al menos un dígito numérico.
  6. Al menos un carácter especial `($, *, -, +, !, ?)`
- Documenta el código fuente con comentarios aclaratorios adicionales.

14.

- Escribe un método que se llame `encripta()` que reciba un texto y una clave y devuelve el texto encriptado en otra cadena. El cifrado debe hacerse aplicando XOR entre cada letra del texto y una letra de la `clave`, cuando se terminan las letras de la clave vuelve a la primera.
- Prueba el método pedido desde `main()`.
- Utiliza la longitud de la clave, `clave.length()`. Para mapear cada carácter del texto a encriptar con un carácter de la clave, calcula el módulo `indice % clave.length()`. Se puede realizar la operación XOR binaria entre caracteres convirtiéndolos en números de tipo `short`.
- Documenta el código fuente con comentarios aclaratorios adicionales.

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en línea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- SÁNCHEZ, J. *Apuntes de fundamentos de programación*. [en línea] http://www.jorgesanchez.net
- GARCÍA HERNÁNDEZ, E. *Ejemplos de Expresiones Regulares en Java*. [en línea] http://puntocomnoesunlenguaje.blogspot.com.es/2013/07/ejemplos-expresiones-regulares-java-split.html
- VOGEL, L. *Java Regex Tutorial*. [en línea] http://www.vogella.com/tutorials/JavaRegularExpressions/article.html
- GOYVAERTS, J. *Regular-Expressions.info.* [en línea] http://www.regular-expressions.info/