

## 1.- Cadenas de caracteres: clase **String**

Probablemente, una de las cosas que más utilizarás cuando estés programando en cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten almacenar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

En Java y en todo lenguaje de programación, y por ende, en todo sistema informático, los caracteres se codifican como secuencias de bits que representan a los símbolos usados en la comunicación humana.

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena. Consiste simplemente en una secuencia de caracteres entre comillas dobles, por ejemplo: "**Ejemplo de cadena de caracteres**".

En Java, los literales de cadena son en realidad instancias de la clase **String**, lo cual quiere decir que, por el mero hecho de escribir un literal, se creará una instancia de dicha clase. Esto da mucha flexibilidad, puesto que permite crear cadenas de muchas formas diferentes, pero obviamente consume mucha memoria. La forma más habitual es crear una cadena partiendo de un literal:

```
String cad="Ejemplo de cadena";
```

En este caso, el literal de cadena situado a la derecha del igual es en realidad una instancia de la clase **String**. Al realizar esta asignación hacemos que la variable **cad** se convierta en una referencia al objeto ya creado. Otra forma de crear una cadena es usando el operador **new** y un constructor, como por ejemplo:

```
String cad=new String ("Ejemplo de cadena");
```

Cuando se crean las cadenas de esta forma, se realiza una copia en memoria de la cadena pasada por parámetro. La nueva instancia de la clase **String** hará referencia por tanto a la copia de la cadena, y no al original.

### 2.1.- Operaciones avanzadas con cadenas de caracteres (I).

¿Qué operaciones puedes hacer con una cadena? Muchas más de las que te imaginas. Empezaremos con la operación más sencilla: la **concatenación**. La concatenación es la unión de dos cadenas, para formar una sola. En Java es muy sencillo, pues sólo tienes que utilizar el operador de concatenación (signo de suma):

```
String cad = "¡Bien" + "venido!";  
System.out.println(cad);
```

En la operación anterior se está creando una nueva cadena, resultado de unir dos cadenas: una cadena con el texto "¡Bien", y otra cadena con el texto "venido!". La segunda línea de código muestra por la salida estándar el resultado de la concatenación. El resultado de su ejecución será que aparecerá el texto "¡Bienvenido!" por la pantalla.

Otra forma de usar la concatenación, que ilustra que cada literal de cadena es a su vez una instancia de la clase **String**, es usando el método **concat** del objeto **String**:

```
String cad="¡Bien".concat("venido!");  
System.out.printf(cad);
```

Fíjate bien en la expresión anterior, pues genera el mismo resultado que la primera opción y en ambas participan tres instancias de la clase **String**. Una instancia que contiene el texto "¡Bien", otra instancia que contiene el texto "venido!", y otra que contiene el texto "¡Bienvenido!". La tercera cadena se crea nueva al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. Pero no te preocupes por las otras dos cadenas, pues se borrarán de memoria cuando el recolector de basura detecte que ya no se usan.

Fíjate además, que se puede invocar directamente un método de la clase **String**, posponiendo el método al literal de cadena. Esto es una señal de que al escribir un literal de cadena, se crea una instancia del objeto **String**.

Pero no solo podemos concatenar una cadena a otra cadena. Gracias al método `toString()` podemos concatenar cadenas con literales numéricos e instancias de otros objetos sin problemas.

El método `toString()` es un método disponible en todas las clases de Java. Su objetivo es simple, permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia. Lo de convertir, no siempre es posible, hay clases fácilmente convertibles a texto, como es la clase `Integer`, por ejemplo, y otras que no se pueden convertir, y que el resultado de invocar el método `toString()` es información relativa a la instancia.

La gran ventaja de la concatenación es que el método `toString()` se invocará automáticamente, sin que tengamos que especificarlo, por ejemplo:

```
Integer i1 = new Integer (1223); // La instancia i1 de la clase Integer contiene el
número 1223.
System.out.println("Número: " + i1); // Se mostrará por pantalla el texto "Número:
1223"
```

En el ejemplo anterior, se ha invocado automáticamente `i1.toString()`, para convertir el número a cadena. Esto se realizará para cualquier instancia de clase concatenada, pero cuidado, como se ha dicho antes, no todas las clases se pueden convertir a cadenas.

### 2.1.1.- Operaciones avanzadas con cadenas de caracteres (II).

Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable `cad` contiene la cadena `"¡Bienvenido!"`, como se muestra en las imágenes.

- `int length()`. Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.

String cad="¡ B i e n v e n i d o !"   
 Longitud = 12

- `char charAt(int pos)`. Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato `char`. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1. Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":

```
char t = cad.charAt(5);
System.out.println(t);
```

String cad="¡ B i e n v e n i d o !"   
 Posición: 0 1 2 3 4 5 6 7 8 9 10 11   
 cad.charAt(0) cad.charAt(4) cad.charAt(11)

- `String substring(int beginIndex, int endIndex)`. Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición `beginIndex` y la posición `endIndex - 1`. Por ejemplo, si pusiéramos `cad.substring(0,5)` en nuestro programa, sobre la variable `cad` anterior, dicho método devolvería la subcadena `"¡Bien"` tal y como se muestra en la imagen.

String cad="¡ B i e n v e n i d o !"   
 Posición: 0 1 2 3 4 5 6 7 8 9 10 11   
 substring(0,5) substring(5,11)   
 String cad="¡ B i e n v e n i d o !"   
 Posición: 0 1 2 3 4 5 6 7 8 9 10 11   
 substring(0)

- `String substring (int beginIndex)`. Cuando al método `substring` solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el carácter con posición `beginIndex` e irá hasta el final de la cadena. En el siguiente ejemplo se mostraría por pantalla la cadena "ienvenido!":

```
String subcad = cad.substring(2);
System.out.println(subcad);
```

### 2.1.3.- Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas? Java ofrece un montón de operaciones más sobre cadenas de caracteres. En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables **cad1**, **cad2** y **cad3** son cadenas ya existentes, y la variable **num** es un número entero mayor o igual a cero.

#### Métodos importantes de la clase String.

Método.	Descripción
<code>cad1.compareTo(cad2)</code>	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena ( <code>cad1</code> ) es anterior en orden alfabético a la que se pasa por argumento ( <code>cad2</code> ), y un número mayor que cero si la cadena es posterior en orden alfabético.
<code>cad1.equals(cad2)</code>	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "==", sino el método <code>equals</code> . Retornará <code>true</code> si son iguales, y <code>false</code> si no lo son.
<code>cad1.compareToIgnoreCase(cad2)</code> <code>cad1.equalsIgnoreCase(cad2)</code>	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
<code>cad1.trim()</code>	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
<code>cad1.toLowerCase()</code>	Genera una copia de la cadena con todos los caracteres a minúscula.
<code>cad1.toUpperCase()</code>	Genera una copia de la cadena con todos los caracteres a mayúsculas.
<code>cad1.indexOf(cad2)</code> <code>cad1.indexOf(cad2, num)</code>	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .

## Métodos importantes de la clase String.

Método.	Descripción
<code>cad1.replace(cad2, cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se reemplazarán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzzz" y no "zzxx".

### 2.1.4.- Operaciones avanzadas con cadenas de caracteres (IV).

¿Sabes cuál es el principal problema de las cadenas de caracteres? Su alto consumo de memoria. Cuando realizamos un programa que realiza muchísimas operaciones con cadenas, es necesario optimizar el uso de memoria.

En Java, **String** es un objeto inmutable, lo cual significa, entre otras cosas, que cada vez que creamos un **String**, o un literal de **String**, se crea un nuevo objeto que no es modificable. Java proporciona la clase **StringBuilder**, la cual es un mutable, y permite una mayor optimización de la memoria. También existe la clase **StringBuffer**, pero consume mayores recursos al estar pensada para aplicaciones multi-hilo, por lo que en nuestro caso nos centraremos en la primera.

Pero, ¿en que se diferencian **StringBuilder** de la clase **String**? Pues básicamente en que la clase **StringBuilder** permite modificar la cadena que contiene, mientras que la clase **String** no. Como ya se dijo antes, al realizar operaciones complejas se crea una nueva instancia de la clase **String**.

Veamos un pequeño ejemplo de uso de esta clase. En el ejemplo que vas a ver, se parte de una cadena con errores, que modificaremos para ir haciéndola legible. Lo primero que tenemos que hacer es crear la instancia de esta clase. Se puede inicializar de muchas formas, por ejemplo, partiendo de un literal de cadena:

```
StringBuilder strb = new StringBuilder ("Hoal Muuundo");
```

Y ahora, usando los métodos **append** (insertar al final), **insert** (insertar una cadena o carácter en una posición específica), **delete** (eliminar los caracteres que hay entre dos posiciones) y **replace** (reemplazar los caracteres que hay entre dos posiciones por otros diferentes), rectificaremos la cadena anterior y la haremos correcta:

1. `strb.delete(6,8)`; Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método `substring`).
2. `strb.append ("!")`; Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. `strb.insert (0, ";")`; Insertamos en la posición 0, el símbolo de apertura de exclamación.
4. `strb.replace (3,5,"la")`; Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos `delete` y `substring`, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

**StringBuilder** contiene muchos métodos de la clase **String** (`charAt`, `indexOf`, `length`, `substring`, `replace`, etc.), pero no todos, pues son clases diferentes con funcionalidades realmente diferentes.

### 2.1.4.- Formateo de cadenas (V): uso método format()

En Java, el método estático **format** de la clase **String** permite formatear los datos que se muestran al usuario o la usuaria de la aplicación. El método **format** tiene los siguientes argumentos:

- **Cadena de formato.** Cadena que especifica cómo será el formato de salida, en ella se mezclará texto normal con especificadores de formato, que indicarán cómo se debe formatear los datos.
- **Lista de argumentos.** Variables que contienen los datos cuyos datos se formatearán. Tiene que haber tantos argumentos como especificadores de formato haya en la cadena de formato.

Los especificadores de formato comienzan siempre por "%", es lo que se denomina un carácter de escape (carácter que sirve para indicar que lo que hay a continuación no es texto normal, sino algo especial). El especificador de formato debe llevar como mínimo el símbolo "%" y un carácter que indica la conversión a realizar, por ejemplo "%d".

La conversión se indica con un simple carácter, y señala al método **format** cómo debe ser formateado el argumento.

Dependiendo del tipo de dato podemos usar unas conversiones u otras. Veamos las conversiones más utilizadas:

#### Listado de conversiones más utilizada y ejemplos.

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Valor lógico o booleano.	"%b" o "%B"	Boolean (cuando se usan otros tipos de datos siempre lo formateará escribiendo true).	<pre>boolean b=true; String d= String.format("Resultado: %b", b); System.out.println (d);</pre>	Resultado: true
Cadena de caracteres.	"%s" o "%S"	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el método toString).	<pre>String cad="hola mundo"; String d= String.format("Resultado: %s", cad); System.out.println (d);</pre>	Resultado: hola mundo
Entero decimal	"%d"	Un tipo de dato entero.	<pre>int i=10; String d= String.format("Resultado: %d", i); System.out.println (d);</pre>	Resultado:10
Número en notación científica	"%e" o "%E"	Flotantes simples o dobles.	<pre>double i=10.5; String d= String.format("Resultado: %E", i); System.out.println (d);</pre>	Resultado:1.0 50000E+01
Número decimal	"%f"	Flotantes simples o dobles.	<pre>float i=10.5f; String d= String.format("Resultado: %f", i); System.out.println (d);</pre>	Resultado: 10,500000

## Listado de conversiones más utilizada y ejemplos.

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Número en notación científica o decimal (lo más corto)	"%g" o "%G"	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea más corto.	<pre>double i=10.5; String d= String.format("Resultado: %g", i); System.out.println (d);</pre>	<p>Resultado:</p> <p>10.5000</p>

Ahora que ya hemos visto alguna de las conversiones existentes (las más importantes), veamos algunos modificadores que se le pueden aplicar a las conversiones, para ajustar como queremos que sea la salida. Los modificadores se sitúan entre el carácter de escape ("%") y la letra que indica el tipo de conversión (d, f, g, etc.).

Podemos especificar, por ejemplo, el número de caracteres que tendrá como mínimo la salida de una conversión. Si el dato mostrado no llega a ese ancho en caracteres, se rellenará con espacios (salvo que se especifique lo contrario):

**%[Ancho]Conversión**

El hecho de que esté entre corchetes [ ] significa que es opcional. Si queremos por ejemplo que la salida genere al menos 5 caracteres (poniendo espacios delante) podríamos ponerlo así:

```
String.format ("%5d",10);
```

Se mostrará el " 10" pero también se añadirán 3 espacios delante para rellenar. Este tipo de modificador se puede usar con cualquier conversión.

Cuando se trata de conversiones de tipo numéricas con decimales, solo para tipos de datos que admitan decimales, podemos indicar también la precisión, que será el número de decimales mínimos que se mostrarán:

**%[Ancho][.Precisión]Conversión**

Como puedes ver, tanto el ancho como la precisión van entre corchetes, los corchetes no hay que ponerlos, solo indican que son modificaciones opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

```
String.format ("%3f", 4.2);
```

Como el número indicado como parámetro solo tiene un decimal, el resultado se completará con ceros por la derecha, generando una cadena como la siguiente: "4,200".

Una cadena de formato puede contener varios especificadores de formato y varios argumentos. Veamos un ejemplo de una cadena con varios especificadores de formato:

```
String np="Lavadora";
int u=10;
float ppu = 302.4f;
float p=u*ppu;
```

```
String output = String.format("Producto: %s; Unidades: %d; Precio por unidad: %.2f €;  
Total: %.2f €", np, u, ppu, p);  
System.out.println(output);
```

Cuando el orden de los argumentos es un poco complicado, porque se reutilizan varias veces en la cadena de formato los mismos argumentos, se puede recurrir a los índices de argumento. Se trata de especificar la posición del argumento a utilizar, indicando la posición del argumento (el primer argumento sería el 1 y no el 0) seguido por el símbolo del dólar ("\$"). El índice se ubicaría al comienzo del especificador de formato, después del porcentaje, por ejemplo:

```
int i=10;  
int j=20;  
String d=String.format("%1$d multiplicado por %2$d (%1$dx%2$d) es %3$d",i,j,i*j);  
System.out.println(d);
```

El ejemplo anterior mostraría por pantalla la cadena **"10 multiplicado por 20 (10x20) es 200"**. Los índices de argumento se pueden usar con todas las conversiones, y es compatible con otros modificadores de formato (incluida la precisión).