Tema 3.-La clase String y sus métodos

Contenido

1 Introducción....2
2 Comparación entre objetos String....5
3 String.valueOf....6
4 Métodos de las variables de tipo cadena....6
4.1 length....6
4.2 concatenar cadenas....6
4.3 charAt....6
4.4 substring....6
4.5 indexOf....6
4.6 lastIndexOf....7
4.7 endsWith....7
4.8 startsWith....7
4.9 replace....7
4.10 replaceAll....7
4.11 toUpperCase....7
4.12 toLowerCase....7
4.13 toCharArray....7
4.14 split....7
5 Lista completa de métodos....8
6 Ejemplo de algunas cosas de lo explicado en este tema....9

1 Introducción

Todos los tipos primitivos de datos tienen un tamaño fijo, conocido de antemano. Por ejemplo,

- sabemos que el tipo **`int`** ocupa 32 bits,
- y que el tipo **`double`** ocupa 64 bits, etc.
- Siguiendo con el tipo **`int`**, sea cual sea el valor **`int`** que tome una variable, sabemos que va a ocupar exactamente 32 bits.

Esta característica es fundamental. Cuando declaramos una variable de un tipo básico, el sistema le asocia una zona de memoria libre donde quepa cualquier dato de ese tipo. En el caso de **`int`**, buscaría una zona de memoria de 32 bits libres, y esa zona la asociaría permanentemente con el nombre de la variable.

Si queremos **cambiar** el valor de la variable basta con acceder a esa posición y escribir el nuevo valor. Sea cual sea, seguro que ocupa exactamente el espacio disponible para la variable.

Esto permite que el compilador **asocie el identificador de la variable con la dirección de memoria** donde se almacena su valor, pudiendo sustituir al compilar todas las alusiones a ese identificador por su dirección de memoria, que siempre va a ser la misma, porque en ella cabe cualquier valor posible de ese tipo de dato.

Para Java las cadenas de texto no son datos de algún tipo de los primitivos sino que son objetos especiales, objetos de la clase **`String`**.

En general, no podemos saber el tamaño exacto que va a ocupar cada objeto de una clase. Por ejemplo, para los objetos de tipo **`String`**, podemos tener valores muy diferentes. Si cada carácter Unicode ocupa 16 bits, cuanto más larga sea una frase, más espacio de memoria necesitará para almacenarse.

Nos podemos preguntar qué espacio debemos reservar para almacenar cada literal o cada objeto de tipo **`String`.** En principio se podría pensar en dos alternativas:

- **Reservar para cualquier variable de tipo `String` un mismo tamaño fijo**, lo suficientemente grande como para que no tengamos problemas al representar cualquier valor. **Inconveniente: en la mayoría de los casos estaríamos desperdiciando mucha memoria, y en algunos pocos casos, ni siquiera sería suficiente la memoria reservada**. En algunos lenguajes, ésta es la opción elegida.
- **Usar objetos**. Es la alternativa usada en Java. **Consiste en que no se reserva ningún espacio para las variables que no sean tipos básicos.** Cuando se declara un objeto, en la zona de memoria que se le asigna no se almacena el valor que nos interesa guardar, sino la dirección de memoria donde habrá que buscar el valor, donde realmente se almacena el valor. **Con esto conseguimos que al compilar la variable se ligue con una dirección fija,** al igual que con los tipos primitivos**, pero en esa posición no buscamos el valor que nos interesa, sino la dirección en la que va a estar ese valor (la referencia). Además ese valor estará ocupando exactamente el espacio que necesita.**

Cualquier otra clase necesita invocar al operador **new** para crear nuevos objetos de esa clase. Sin embargo, la clase **`String`** se usa tanto, que los diseñadores del lenguaje incluyeron una versión abreviada de esa llamada al constructor.

Por eso hay **dos formas** de crear objetos de tipo **`String`**:

- Como si de un tipo primitivo se tratase. Esto es, directamente asignándole un literal de tipo String. `String texto = "Pepe";`
- Invocando a alguno de los constructores de la clase. Esto es, usando el operador new. `String texto = new String("Pepe");`

Las dos sentencias anteriores son, a todos los efectos, equivalentes

Posteriormente podemos cambiar el valor de texto:

```
texto= "Nicomedes Gumersindo Iraolagoitia Zumalacandarría"
```

Las cadenas pueden concatenar varias cadenas utilizando el operador de concatenación “+”.

```
String texto2 ="Este es un texto que ocupa " +
               "varias líneas, no obstante se puede " +
               "perfectamente encadenar";
```

Las referencias funcionan de la siguiente forma:

- Cuando ejecutamos la primera sentencia en la que **`texto`** toma el valor **"Pepe"**, **al declarar esa variable**, se busca una zona de memoria en la que quepa justamente eso, una dirección de memoria (**las direcciones de memoria siempre ocupan el mismo tamaño**, a fin de cuentas son números). Supongamos que la encuentra en la dirección de memoria 3000.
- Etiquetamos la dirección encontrada con el nombre **`texto`**, y eso ya no va a cambiar. Siempre que en el programa aparezca el identificador **`texto`**, el compilador lo va a sustituir por la dirección 3000, para que el programa acceda a esa posición en busca del valor de texto.
- Cuando el constructor **`new`** crea el nuevo objeto **`String`**, **busca espacio libre en memoria donde poder alojar cuatro caracteres, los necesarios para el literal `String` "Pepe"**. Supongamos que lo encuentra en la dirección de memoria 5000.
- En esa posición crearía el objeto, de forma que la zona de memoria que comienza en la posición 5000 contendrá el literal "Pepe".
- Pero necesitamos ligar el identificador **`texto`** con el valor **"Pepe"**, por lo que **en la dirección 3000 lo que guardamos realmente es el valor 5000**. El programa al ejecutarse accederá a la posición 3000 y al encontrar allí almacenada la dirección 5000, accederá a ésta en busca del valor para **texto**.
- Cuando posteriormente cambiamos el valor de **`texto`** por otro más largo que ocupa 784 bits, ("Nicomedes Gumersindo Iraolagoitia Zumalacandarría") buscamos un nuevo trozo de memoria libre donde quepa el nuevo valor. Supongamos que lo encuentra en la dirección de memoria 6000.
- Guarda empezando en la posición 6000 el nuevo valor, "Nicomedes Gumersindo Iraolagoitia Zumalacandarría", que ocupa los 784 bits que necesita.
- Actualiza la referencia, es decir, guarda el valor 6000 en la posición 3000. De esta manera, cuando encontremos en el programa el identificador **`texto`**, iremos a su dirección asociada, que es la 3000, y veremos que contiene el valor 6000, que es la dirección a la que iremos a buscar el valor actual para **`texto`**.
- Cualquier nueva actualización que modifique el tamaño del objeto, alojará ese nuevo objeto modificado en una nueva zona de memoria libre y actualizará la referencia, pero nos da lo mismo la dirección concreta que sea. El programa siempre irá a consultar la dirección en la que está el objeto a la dirección 3000.
- **Por tanto, no es necesario limitar el tamaño. Cuando cambia el objeto, se busca un nuevo espacio en memoria donde poder alojarlo de forma que ocupe sólo el espacio que necesita, y actualizamos el valor en la referencia.**

¿Qué pasa con el valor anterior, es decir, con el objeto antiguo, una vez que actualizamos la referencia?

Realmente no lo hemos borrado de la memoria. Siguiendo con el ejemplo anterior, cuando en la posición 3000 cambiamos el valor de 5000 a 6000, no hemos cambiado nada en la posición 5000, que sigue conteniendo el mismo valor que tenía. Si no nos ocupamos de esto, la memoria se va a llenar con valores viejos, que ya no utilizamos.

Efectivamente, habrá que liberar el espacio de memoria que ocupaban esos viejos objetos, que ya no son útiles, dejándolo disponible para cualquier otro propósito.

En algunos lenguajes, como C, es el programador el que debe tener cuidado con destruir esos objetos, liberando la memoria que ocupaban. Y si no haces las cosas bien en esos lenguajes, puedes haber dejado un objeto desconectado, sin ninguna referencia, lo que haría que ya tampoco pudiéramos llegar a él para borrarlo.

Afortunadamente, las cosas en Java son bastante más fáciles.

Nosotros lo único que hacemos es actualizar la referencia, de forma que no queda ninguna referencia que apunte al objeto antiguo. Eso consigue eliminar el objeto antiguo:

- **Java dispone de un programa de baja prioridad que se ejecuta en segundo plano, y que mantiene una lista actualizada de todas las referencias.**
- **Cuando encuentra una dirección de memoria ocupada a la que no apunta ninguna referencia, la identifica como basura, y la libera.**
- **Si no hay ninguna referencia, no hay forma de acceder a ese objeto, por lo que a todos los efectos, es imposible volver a usarlo. Así que realmente es adecuado considerarlo como basura y eliminarlo automáticamente.**
- Ese programa es **el recolector automático de basura** (**garbage collector**).
- **El recolector de basura se activa cuando la máquina virtual Java lo estima oportuno**, sin que nosotros intervengamos para nada.
- De hecho, **aunque existe un método gc() para activar el recolector de basura** cuando detectamos que se está produciendo mucha basura en nuestro programa, **no es más que una recomendación a tener en cuenta por la máquina virtual, que no garantiza para nada que el recolector de basura se active precisamente en ese momento**.

La existencia del recolector automático de basura **permite evitar el uso de punteros que tanto complica la programación en otros lenguajes**.

A cambio de esa facilidad, se puede perder algo de eficiencia, ya que puedes tener en algunos momentos mucha memoria ocupada inútilmente por basura, a la espera de que se active el recolector de basura. Además, la comprobación de las referencias para ver si una zona ocupada de memoria es o no basura, también consume un cierto tiempo de procesamiento, incidiendo en la eficiencia. Nada es gratis, pero en este caso el precio merece la pena. Además, esa pérdida de eficiencia es perfectamente asumible si tenemos en cuenta la potencia de procesamiento de los ordenadores actuales.

También se pueden crear objetos `String` sin utilizar constantes entrecomilladas, usando otros constructores:

```
char[] palabra = {'P','a','l','b','r','a'};//Array de char
String cadena = new String(palabra);
```

Antes de trabajar con `String` hay que hacer una parada con el tipo primitivo `char` que ya vimos en la declaración de dichos tipos. El tipo `char` ocupa dos bytes y es perfectamente reversible con los `int`, es decir, se puede asignar un `int` a un `char` y viceversa. El valor que toma el `int` es el código ASCII correspondiente a dicho carácter, para ver la tabla de códigos ASCII consultar:

http://www.elcodigoascii.com.ar/

y en el caso de asignar un numero a un `char` toma el carácter de ese código siempre y cuando esté entre 0-255. De todas formas es mejor trabajar siempre con códigos desde el 0 al 128 pues del 128 en adelante es el ASCII extendido particularizado para cada idioma. Fuera del rango 0-255 el resultado es impredecible.

Ejemplo:

```
int t;
t='A';//(los char siempre con comillas simples los String con comillas
       //dobles)
System.out.println("valor de t:"+t);
```

Esto nos devolverá un 65 por pantalla que es el código de la A mayúscula

Ejemplo:

```
char letra;
letra=97;
System.out.println("valor de letra:"+letra);
```

Esto devolverá una a minúscula que es el código de correspondiente al 97

Otra peculiaridad de la clase `String` es que genera **objetos inmutables**, es decir, que no se pueden modificar desde que se crean hasta que se destruyen**.**

¿Qué quiere decir esto? Aparentemente podemos modificarlos. Si yo escribo:

```
String a = "Hola"
a = a + " caracola";//El operador + para String concatena las cadenas.
System.out.println(a);
```

El resultado es un mensaje escrito que dice:

```
Hola caracola
```

Aparentemente sí que se ha modificado el valor de la variable **a**, no parece inmutable. Pero sólo es aparentemente. Lo que ocurre en realidad es que:

- **cada vez que hacemos alguna modificación sobre el objeto `String`,**
- **se crea un objeto nuevo sobre el anterior modificado,**
- **y se actualiza la referencia**.

Con la misma referencia **a** llego al nuevo objeto, y por eso parece que lo he modificado, pero lo que he hecho es crear otro nuevo. Mientras, el viejo objeto será tarde o temprano eliminado de la memoria por el recolector de basura.

2 Comparación entre objetos String

Los objetos **String** no pueden compararse directamente con los operadores de comparación. En su lugar se deben utilizar estas expresiones:

- *`cadena1.equals(cadena2)`* **.** El resultado es **true** si la cadena1 es igual a la cadena2. Ambas cadenas son variables de tipo **String.**
- *`cadena1.equalsIgnoreCase(cadena2)`* **.** Como la anterior, pero en este caso no se tienen en cuenta mayúsculas y minúsculas.
- *`s1.compareTo(s2)`* **.** Compara ambas cadenas, considerando el orden alfabético.

Si la primera cadena es mayor en orden alfabético que la segunda devuelve un entero **mayor que 0**, si son iguales devuelve 0 y si es la segunda la mayor devuelve un entero **menor que 0**. Hay que tener en cuenta que el orden no es el del alfabeto español, sino que usa la tabla ASCII, en esa tabla la letra *ñ* es mucho mayor que la *o*.

- *`s1.compareToIgnoreCase(s2)`*. Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

3 String.valueOf

Este método convierte valores de una clase a otra. En el caso de los objetos String, permite convertir valores que no son de cadena a forma de cadena. Ejemplos:

```
String numero = String.valueOf(1234);
```

4 Métodos de las variables de tipo cadena

Son métodos que poseen las propias variables de cadena. Para utilizarlos basta con poner el nombre de la variable String seguido de un punto (.) y el nombre del método y sus parámetros. **La variable con la que se invoca el método no puede ser *null***:

```
variableString.método(argumentos)
```

4.1 length

Permite devolver la longitud de una cadena (el número de caracteres de la cadena):

```
String texto1 = "Prueba";
System.out.println(texto1.length());//Escribe 6
```

4.2 concatenar cadenas

Se puede hacer de dos formas, utilizando el método **concat** o con el operador +.

Ejemplo:

```
String s1 = "Buenos ", s2 = "días", s3, s4;
s3 = s1 + s2;
s4 = s1.concat(s2);//equivalentes
```

4.3 charAt

Devuelve un carácter de la cadena. El carácter a devolver se indica por su posición (el primer carácter es la posición 0) Si la posición es negativa o sobrepasa el tamaño de la cadena, ocurre un error de ejecución, una excepción tipo **`IndexOutOfBounds-Exception`**. Ejemplo:

```
String s1 = "Prueba";
char c1 = s1.charAt(2); //c1 valdrá 'u'
```

4.4 substring

Da como resultado una porción del texto de la cadena. La porción se toma desde una posición inicial hasta una posición final (sin incluir esa posición final). Si las posiciones indicadas no son válidas ocurre una excepción de tipo **`IndexOutOfBounds-Exception`.** Se empieza a contar desde la posición 0. Ejemplo:

```
String s1 = "Buenos días";
String s2 = s1.substring(7,10); //s2 = día
```

4.5 indexOf

Devuelve la primera posición en la que aparece un determinado texto en la cadena. En el caso de que la cadena buscada no se encuentre, devuelve -1. El texto a buscar puede ser **char** o **String.** Ejemplo:

```
String s1 = "Quería decirte que quiero que te vayas";
System.out.println(s1.indexOf("que")); //Da 15
```

Se puede buscar desde una determinada posición. En el ejemplo anterior:

```
System.out.println(s1.indexOf("que",16)); //Ahora da 26
```

4.6 lastIndexOf

Devuelve la última posición en la que aparece un determinado texto en la cadena. Es casi idéntica a la anterior, sólo que busca desde el final. Ejemplo:

```
String s1 = "Quería decirte que quiero que te vayas";
System.out.println(s1.lastIndexOf("que"); //Da 26
```

También permite comenzar a buscar desde una determinada posición.

4.7 endsWith

Devuelve **true** si la cadena termina con un determinado texto. Ejemplo:

```
String s1 = "Quería decirte que quiero que te vayas";
System.out.println(s1.endsWith("vayas"); //Da true
```

4.8 startsWith

Devuelve **true** si la cadena empieza con un determinado texto.

4.9 replace

Cambia todas las apariciones de un carácter por otro en el texto que se indique y lo almacena como resultado. El texto original no se cambia, por lo que hay que asignar el resultado de **replace** a un String para almacenar el texto cambiado:

```
String s1 = "Mariposa";
System.out.println(s1.replace('a', 'e'));//Da Meripose
System.out.println(s1);//Sigue valiendo Mariposa
```

4.10 replaceAll

Modifica en un texto cada entrada de una cadena por otra y devuelve el resultado. El primer parámetro es el texto que se busca (que puede ser una expresión regular), el segundo parámetro es el texto con el que se reemplaza el buscado. La cadena original no se modifica.

```
String s1 = "Cazar armadillos";
System.out.println(s1.replaceAll("ar","er"));//Da Cazer ermedillos
System.out.println(s1);//Sigue valiendo Cazar armadillos
```

4.11 toUpperCase

Devuelve la versión en mayúsculas de la cadena. No cambia la cadena original.

4.12 toLowerCase

Devuelve la versión en minúsculas de la cadena. No cambia la cadena original.

4.13 toCharArray

Obtiene un array de caracteres a partir de una cadena.

4.14 split

Devuelve un vector de String en el que cada elemento se forma con el trozo de la cadena cada vez que encuentra un valor de la expresión regular en la cadena que invoca el método. La expresión regular se pasa como parámetro al método.

Al final de este tema hay un ejemplo con el uso de algunos métodos de la clase String (no todos) y la reversibilidad de char a int.

5 Lista completa de métodos.

Result.	Método	Descripción
char	**charAt(**int index**)**	Proporciona el carácter que está en la posición dada por el entero *index* de la cadena que lo invoca (empieza en 0).
int	**compareTo(**String s**)**	Compara las dos cadenas alfabéticamente (ASCII). Devuelve un valor **menor que cero** si la cadena que invoca el método es menor que *s*, devuelve 0 si son iguales y devuelve un valor **mayor que cero** si la cadena que invoca al método es mayor que *s*.
int	**compareToIgnoreCase(**String s)	Compara dos cadenas como **compareTo**, pero no tiene en cuenta si el texto es mayúsculas o no.
String	**concat(**String s**)**	Añade la cadena *s* al final de la cadena que llama al método, pero sin que se actualice la original.
String **(static)**	**String.copyValueOf(**char[] data**)**	Produce un objeto String que es igual al array de caracteres data.
boolean	**endsWith(**String s**)**	Devuelve **true** si la cadena termina con el texto s.
boolean	**equals(**String s**)**	Compara ambas cadenas, devuelve **true** si son iguales, sino **false.**
boolean	**equalsIgnoreCase(**String s**)**	Compara ambas cadenas como **equals** sin tener en cuenta las mayúsculas y las minúsculas.
String **(static)**	**String.format(**String f, Objeto o, ..)	Devuelve un String con el formato fijado por **f** de/los objeto/s o. Tiene que haber el mismo número de argumentos que elementos estén referidos en el formato.
byte[]	**getBytes()**	Devuelve un array de caracteres que toma a partir de la cadena que lo invoca.
void	**getBytes(**int srcBegin, int srcEnd, char[] dest, int dstBegin);	Almacena el contenido de la cadena en el array de caracteres *dest*. Toma los caracteres desde la posición *srcBegin* hasta la posición *srcEnd* y los copia en el array desde la posición *dstBegin.*
int	**indexOf(**String s**)**	Devuelve la posición del texto s en la cadena que lo invoca. Si no lo encuentra devuelve -1.
int	**indexOf(**String s, int primeraPos)	Como **indexOf** anterior, devuelve la posición del texto *s* en la cadena que lo invoca, pero empezando a buscar desde la posición *primeraPos.*
boolean	**isEmpty()**	Devuelve **true** solo si la longitud de la cadena que lo invoca es 0.
int	**lastIndexOf(**String s**)**	Devuelve la última posición en que aparece la cadena del texto s en la cadena que lo invoca.
int	**lastIndexOf(**String s, int primeraPos)	Devuelve la última posición del texto *s* en la cadena que lo invoca, empezando a buscar desde la posición *primeraPos.*
int	**length()**	Devuelve la longitud de la cadena que lo invoca.
boolean	**match(**String regex)	Devuelve **true** si la cadena que lo invoca cumple la expresión regular, **false** si no la cumple
String	**replace(**char carAnterior, char carNuevo)	Devuelve una cadena idéntica a la original en la que ha cambiando los caracteres iguales a *carAnterior* por *carNuevo.* La cadena original no se modifica. También produce el cambio si los parámetros son una secuencia de caracteres.
String	**replaceAll(**String regex, String str2**)**	Cambia la todas las apariciones de la de la expresión regular *regex* (puede ser una cadena) por la cadena *str2*. La original no cambia.
String	**replaceFirst(**String regex, String str2**)**	Cambia la primera aparición de la expresión regular *regex* (puede ser una cadena) por la cadena *str2*. La original no se modifica.
boolean	**startsWith(**String s**)**	Devuelve **true** si la cadena que lo invoca comienza con el texto *s*.
String	**substring(**int primeraPos, intsegundaPos**)**	Devuelve el texto que va desde *primeraPos* a *segundaPos*. Empieza en 0 y *segundaPos* no entra.
char[]	**toCharArray()**	Devuelve un array de caracteres a partir de la cadena dada.
String	**toLowerCase()**	Convierte la cadena a minúsculas. La original no se modifica.

Result.	Método	Descripción
String	**toUpperCase()**	Convierte la cadena a mayúsculas. La original no se modifica.
String	**trim()**	Elimina los blancos que tenga la cadena tanto al principio delante como al final, sin modificar la cadena que lo invoca.
String[]	**split(**String regex**)**	Separa en trozos la cadena que lo invoca. Los trozos se generan cada vez que se encuentra un elemento de la expresión regular.
String **(static)**	**String.valueOf(***tipo* elemento**)**	Devuelve la cadena con el contenido de elemento. Si elemento es booleano, devolvería una cadena con el valor true o false.

6 Ejemplo de algunas cosas de lo explicado en este tema

```
public class Funciones {

	public static void main(String[] args) {
		int t;
		String num ="0123456789", trozo;
		String let ="abcdefghijklmnop";
		char letra;
		//escritura acumulada
		for (t=0;t<=num.length();t++)
		{
			trozo=num.substring(0, t);
			System.out.println("este es el trozo obtenido;"+trozo);
		}
		//deletreo o recorrido de un String carácter a carácter
		for (t=0;t<num.length();t++)
		{
			trozo=num.substring(t, t+1);
			System.out.println("este es el trozo obtenido;"+trozo);
		}

		let=let.replace('c','X');
		let=let.replace('d','X');
		let=let.replace('e','X');
		System.out.println("después del reemplazo de las equis>>>"+let);
		// si solo se ejecuta el método replace el contenido de let no cambia
		let.replace('c','Z');
		let.replace('d','Z');
		let.replace('e','Z');
		System.out.println("después del reemplazo de las zetas>>>"+let);

		letra=num.charAt(0);
		t=letra;
		System.out.println("la letras sexta ha sido"+t);
		let=String.valueOf(letra);
		// Alfabeto mayúsculas
		for (letra=65;letra<=90;letra++)
		System.out.print(letra+"/");
		System.out.println();
		// Alfabeto minúsculas
		for (letra=97;letra<=122;letra++)
		System.out.print(letra+"/");
		// números tratados como caracteres
		for (letra=48;letra<=57;letra++)
			System.out.print(letra+"/");
		System.out.println();

	}// fin de main
}//fin de class
```