BAILADEV

JAVA: STRING MÉTODOS

MÉTODO CONSTRUCTOR *(creación de un objeto de tipo cadena, llamando a la clase String)*

- **String** miCadena = **new String("Mi cadena")**;

EXCEPCIONES: Si nuestra variable de tipo **String** almacena un valor **null** y queremos hacer uso de algunos métodos propuestos en este documento, nos lanzará una excepción de tipo → **NullPointerException** proveniente de la clase **NullPointerException** que se encuentra en el paquete **java** subpaquete **lang** tal que el acceso completo al documento de la excepción es → **java.lang.NullPointerException**.

TIPO DE RETORNO: STRING

MÉTODO → concat()

Descripción

- Este método nos servirá para concatenar/unir una cadena con otra, devolviendo así una nueva cadena que incluye esa unión. Es equivalente a la concatenación que hacemos con el operador "+" en la mayoría de lenguajes.

FÓRMULA DEL MÉTODO: ***miCadena*.concat("*cadena*")**

USO del MÉTODO

EJEMPLO VÁLIDO → "Hola".concat("concatenados") = "Hola concatenado" ← Devuelve
EJEMPLO VÁLIDO → "Hola".concat(" mun" + "do") = "Hola mundo" ← Devuelve

Argumentos: 1-requerido (tipo cadena) → Espera recibir como argumento una segunda cadena a concatenar con la cadena base.

MÉTODO → copyValueOf()

Descripción

- Este método nos servirá para generar una especie de conversión de tipo entre un arreglo de caracteres y una cadena, lo que hará dicho método será tomar nuestro arreglo de caracteres y transformarlo en una cadena completamente pura; como se puede ver, este método es un opuesto al método **toCharArray()**, que toma una cadena y la convierte en arreglo de caracteres.

FÓRMULA DEL MÉTODO: ***miCadena*.copyValueOf(arrChar, posInicial, posFinal)**

USO del MÉTODO → Suponiendo lo siguiente: **char[] caracteres= {'h','o','l','a'};**

EJEMPLO VÁLIDO → "".copyValueOf(caracteres) = "hola" ← Devuelve
EJEMPLO VÁLIDO → "".copyValueOf(caracteres, 0, 2) = "hol" ← Devuelve
EJEMPLO VÁLIDO → "".copyValueOf(caracteres, 0, caracteres.length) = "hola" ← Devuelve

Argumentos: 1-requerido (tipo cadena) → Espera recibir como argumento nuestro arreglo base de caracteres, que dependeremos de este para comenzar a formar nuestra cadena; si solo dejamos este, que es requerido, nos mostrará la cadena completa conforme al conjunto de caracteres tengamos. 2-opcional (tipo entero) → Espera recibir como argumento la posición/índice del elemento "**INICIAL**" que queremos tomar como punto de partida para la formación de nuestra cadena. 3-requerido (tipo entero) → Espera recibir como argumento la posición/índice del elemento "**FINAL**" que queremos tomar como punto de cierre para la formación de nuestra cadena; si hacemos uso del argumento 2, también debemos hacer uso de este argumento, ya que es necesario contar con el límite de nuestro arreglo para este labor.

MÉTODO → toLowerCase()

Descripción

- Este método se encargará de devolvernos una nueva cadena en formato minúscula, es decir, reemplazará cada carácter que se encuentre en mayúscula por caracteres minúsculos a partir de una cadena base. Su resultado será toda la cadena en minúscula.

FÓRMULA DEL MÉTODO: ***miCadena*.toLowerCase()**

USO del MÉTODO

EJEMPLO VÁLIDO → "HOla".toLowerCase() = "hola" ← Devuelve
EJEMPLO VÁLIDO → "HOLA".toLowerCase() = "hola" ← Devuelve

MÉTODO → toUpperCase()

Descripción

- Este método se encargará de devolvernos una nueva cadena en formato mayúscula, es decir, reemplazará cada carácter que se encuentre en minúscula por caracteres mayúsculos a partir de una cadena base. Su resultado será toda la cadena en mayúscula.

FÓRMULA DEL MÉTODO: ***miCadena*.toUpperCase()**

USO del MÉTODO

EJEMPLO VÁLIDO → "HOla".toUpperCase() = "HOLA" ← Devuelve
EJEMPLO VÁLIDO → "hola". toUpperCase() = "HOLA" ← Devuelve

MÉTODO → substring()

Descripción

- Este método nos servirá para traernos las partes deseadas de una cadena, donde nos basaremos en la posición inicial especificada para comenzar a traer el resto de la cadena, aunque el resultado obtenido es una cadena, nosotros traeríamos los caracteres, por medio de la especificación de índices utilizada en los argumentos de este método.
  Este método admite dos valores, donde vamos a especificarle desde qué posición queremos comenzar a visualizar la cadena hasta dónde queremos que termine dicha visualización.

FÓRMULA DEL MÉTODO: ***miCadena*.substring(posInicial, posFinal) | *miCadena*.substring(posInicial)**

USO del MÉTODO

EJEMPLO VÁLIDO → "hola".substring(0,3) = "hol" ← Devuelve
EJEMPLO VÁLIDO → "hola". substring(1) = "ola" ← Devuelve

Argumentos: 1-requerido (tipo entero) → Espera recibir como argumento una entero, este argumento indica la posición "**INICIAL**" donde comenzará a visualizarse la cadena. 2-opcional (tipo entero) → Espera recibir como argumento un entero, este argumento indica la posición "**FINAL**", donde terminará el carácter a visualizar.

NOTA: Si omitimos el segundo argumento, procederá a mostrarnos hasta finalizar la cadena tomando como punto de partida la posición inicial que le hemos especificado en el primer argumento. Si no le colocamos valor a nuestro método nos devolverá la cadena completa.

MÉTODO → trim()

Descripción

- Este método se encargará de eliminar los espacios en blancos que nuestra cadena tenga, tanto al comienzo de esta como al final, devolviendo así una nueva cadena sin espacios en los costados. Este método es ideal si de evaluar igualdad sobre una cadena se trata, en especial cuando ingresamos datos por teclado, ya que tal vez se nos escape un espacio al principio o al final y por culpa de ese pequeño percance tengamos un error de comparación.

FÓRMULA DEL MÉTODO: ***miCadena*.trim()**

USO del MÉTODO

EJEMPLO VÁLIDO → " Hola ".trim() = "Hola" ← Devuelve

MÉTODO → intern()

Descripción

- Este método nos servirá para "**colocar en memoria**" una variable con otra variable o grupo de variables que posea entre ellas el mismo contenido; por lo que para decirle que van a estar compartiendo mismo espacio en memoria, debemos hacer uso de este método, si ambas variables cuentan con la misma información/contenido será **true**, por lo que entonces compartirán mismo espacio en memoria, de lo contrario, si es **false**, no lo harán.
  La creación de nuevos objetos puede resultar una operación un poco costosa, por lo que los desarrolladores de Java para ahorrar tiempo se les han ocurrido la idea de crear un nuevo concepto denominado **String Constant Pool (SCP)**, este consiste en alojarse dentro de la memoria y reservar espacio para colocar aquellas cadenas que son iguales, por lo que esta zona agrupará las cadenas en ese sitio, pero para alojar en este sitio nuestra información debemos hacer uso del método **intern()**, donde solo se creará una única vez un objeto y luego si hay coincidencia con otros comparte referencia tomando como base este objeto que ha sido creado, por lo que no creará más objetos para solo almacenar la misma información**.**
  Antes de proceder con el proceso de agrupamiento, en el momento de compilación, el **JVM** solo le bastará con analizar dicha variable y devolver la referencia en caso de existir igualdad con otra de las variables que se encuentren dentro del grupo, este grupo **SCP** tendrá un código de referencia que compartirá con aquellas cadenas que compartan igualdad.

FÓRMULA DEL MÉTODO: ***miCadena*****.intern()**

USO del MÉTODO

EJEMPLO VÁLIDO → *Sin el método intern() se crearán dos nuevos objetos con espacios individuales.*

```
String cad1=new String("Hola"); ← CREACIÓN DE UN OBJETO
String cad1=new String("Hola"); ← CREACIÓN DE UN OBJETO

System.out.println(cad1 == cad2); ← SERÁ FALSE; no se encuentra compartiendo espacio
```

EJEMPLO VÁLIDO → *Con el método intern() se crearán un objeto que compartirá con otros objetos que comparta igualdad.*

```
String cad1=new String("Hola").intern(); ← CREACIÓN DE UN OBJETO
String cad1=new String("Hola").intern(); ← cad1 comparte referencia en cao de igualdad

System.out.println(cad1 == cad2); ← SERÁ TRUE;  se encuentran compartiendo espacio
```

CONSEJO: Para saber si ambas variables comparten espacio en memoria podemos tomar ambas variables y evaluarlas por medio del operador de igualdad **==**.

NOTA: Si no hacemos uso de este método y ambas variables cuentan con la misma información/contenido, estaríamos creando dos celdas para cada variable. Por lo que el método **intern()** nos provee la solución de compartir misma celda para aquellas variables que cuentan con misma información.

MÉTODO ESTÁTICO → format()

Descripción

- Este método es una nueva forma de trabajar al momento de estar concatenando una cadena con un variable, el término usado para este método se lo conoce como formateo de cadena, es decir que da una apariencia distinta a la conocida para visualizar la información incluyendo el valor de una variable.
  Este método es similar al método **printf()** que trae consigo el mismo Java; la diferencia que hay entre **printf()** y **format()** es que muestra el resultado por consola de la misma forma, fuera de lo que sería el método **println()**, pero estos tres métodos hacen más de lo mismo, por lo que no hay diferencias destacables, más que la forma distinta de hacer uso la concatenación entre la información que pueda poseer una variable y un mensaje/cadena.
  Para hacer uso de este método debemos hacer uso de algo llamado "**ESPECIFICADOR**" que será el encargado de trabajar codo a codo con la información, dependiendo de su tipo de dato. El especificador debe ir dentro de una cadena y al hacer mención de algunos de los especificadores disponibles, debemos anteponer el símbolo porcentaje **(%)** → te invito a que veas el archivo **PDF** "**01_String Formateador**", ahí podrás encontrar una tabla útil, con todos los especificadores disponibles, para qué sirven y para trabajar dependiendo del tipo de dato.
  Al estar empleando este método nos podemos ligar un poco de lo que es la concatenación con el operador "**+**", ya que a la larga puede resultar un poco excesivo su uso.
  Cabe mencionar lo siguiente, cuando usamos el especificador, estaremos reservando lugar para la llamada a la variable, por lo que el especificador usado mostrará la información de esa variable, sería algo así como una variable local propia de la cadena, donde almacenará la información correspondiente y la mostrará (todo en uno).

FÓRMULA DEL MÉTODO: ***String*.format("cadena%ESPECIFICADOR", variable)**

USO del MÉTODO → *es un método estático por lo que debemos utilizar la clase String*
EJEMPLO VÁLIDO → String.format("Hola %s", " Gastón") = "Hola Gastón" ← Devuelve
EJEMPLO VÁLIDO → String.format(" Gastón" , "Hola %s") = "Gastón Hola" ← Devuelve

Argumentos: **1-requerido (tipo cadena, donde se puede incluir uno o más especificadores)** → Espera recibir como argumento una cadena, donde debe venir si o si incluido un especificador, ya que el especificador será el encargado de capturar la información de la variable, si nuestra cadena no cuenta con un especificador dicha información no será mostrada, por más que tenga la variable en el segundo argumento. **2-requerido (variable o cadena)** → Espera recibir como argumento una variable, donde debe contener información para que pueda trabajar a la par con el especificador utilizado en nuestra cadena; si no hacemos uso de este argumento la información en nuestra cadena no será mostrada.

MÉTODO → replaceAll()

Descripción

- Este método hace lo mismo que el método **replace()**, salvo que además de reemplazar todas las coincidencias, también admite **REGEX**, esto quiere decir que podemos por medio de expresiones regulares ser más específicos con los datos que queremos cambiar.
  Este método no funcionará para reemplazar carácter, por lo que si queremos reemplazar carácter, al tratarse de que este reemplaza todas las coincidencias, podemos usar tranquilamente la comillas dobles e introducir un único carácter, el resultado es efectivo.

FÓRMULA DEL MÉTODO: ***miCadena*.replaceAll("cadena", "info_nueva")**

USO del MÉTODO
EJEMPLO VÁLIDO → "Holao".replaceAll("ola" , "corta") = "Hcortao" ← Devuelve
EJEMPLO VÁLIDO → "Holaola".replaceAll("ola" , "corta") = "Hcortaola" ← Devuelve

Argumentos: **Lleva 2 argumentos** y la descripción es la misma que se ha puesto en el método **replace()**, excepto que en este método no podremos utilizar las comillas simples.

NOTA: La Nota es la misma que se ha puesto en el método **replace()**.

MÉTODO → replace()

Descripción

- Este método se encargará de devolvernos una nueva cadena con el reemplazo aplicado, tomando como base el criterio de búsqueda, puede ser un carácter o una cadena el dato a suplantar.
  Este método lo que hace será reemplazar "**globalmente**" (en caso de haber más de una coincidencia con el criterio de búsqueda) toda coincidencia y la suplanta por la que nosotros hayamos introducido en su segundo argumento, logrando así generar una nueva cadena con esos datos aplicados.

FÓRMULA DEL MÉTODO: ***miCadena*.replace("cadena" | 'carácter',"info_nuevo" | 'carácter_nuevo')**

USO del MÉTODO → *es un método estático por lo que debemos utilizar la clase String*
EJEMPLO VÁLIDO → "Holao".replace('o','A') = "HAlaA" ← Devuelve
EJEMPLO VÁLIDO → "Holao".replace("ola" , "corta") = "Hcortao" ← Devuelve
EJEMPLO VÁLIDO → "Holaola".replace("ola" , "corta") = "Hcortacorta" ← Devuelve

Argumentos: **1-requerido (tipo cadena | tipo carácter)** → Espera recibir como argumento una cadena o un carácter existente que se encuentre en nuestra cadena interesados en modificar, en caso de no existir, se nos retornará la cadena exacta sin ninguna modificación aplicada. **2-requerido (tipo cadena | tipo carácter)** → Espera recibir como argumento una cadena o un carácter dependiendo de lo que se haya introducido en el primer argumento, donde vamos a poner la nueva información que queremos que suplante/reemplace a la información vieja expresada en el primer argumento.

NOTA: El criterio de búsqueda puede ser un carácter si solo introducimos un único carácter o una cadena, no olvidarse de las comillas (**simples=>carácter | dobles=>cadena**), dependiendo de si es un carácter o una cadena. El método **replace()** también hay que saber que puede haber una o más existencia en base al criterio de búsqueda y el reemplazo con este método puede afectar a todos los que coincida con el criterio de búsqueda.

MÉTODO → replaceFirst()

Descripción

- Este método hace lo mismo que el método **replace()**, salvo que en lugar de reemplazar todas las coincidencias en base al criterio de búsqueda, solo reemplazará la primera coincidencia y este método no funcionará para reemplazar carácter, por lo que si queremos reemplazar carácter, al tratarse de que este reemplaza la primer coincidencia, podemos usar tranquilamente la comillas dobles e introducir un único carácter, el resultado es efectivo. La diferencia entre este método y **replace()** es que este método admite **REGEX**, esto quiere decir que podemos por medio de expresiones regulares ser más específicos con los datos que queremos cambiar.

FÓRMULA DEL MÉTODO: ***miCadena*.replaceFirst("cadena", "info_nueva")**

USO del MÉTODO → *es un método estático por lo que debemos utilizar la clase String*
EJEMPLO VÁLIDO → "Holao".replaceFirst('o','A') = "HAlao" ← Devuelve
EJEMPLO VÁLIDO → "Holao".replaceFirst("ola" , "corta") = "Hcortao" ← Devuelve
EJEMPLO VÁLIDO → "Holaola".replaceFirst("ola" , "corta") = "Hcortaola" ← Devuelve

Argumentos: **Lleva 2 argumentos** y la descripción es la misma que se ha puesto en el método **replace()**, excepto que en este método no podremos utilizar las comillas simples.

NOTA: La Nota es la misma que se ha puesto en el método **replace()**.

MÉTODO ESTÁTICO → toString()

Descripción

- Este método se encargará de devolvernos cualquier objeto Java a un formato legible de tipo **String**, en pocas palabras convierte a **String** cualquier objeto; todos los objetos del lenguaje, inclusive el propio objeto que hemos creado a partir de nuestra clase, todo objeto en Java dispone de un método **toString()**, ya que este método pertenece a la clase **Object** y todos los objetos son accesibles a ella ya que la clase **Object** es la clase padre de todos los objetos empleados en Java y no necesita utilizar la palabra reservada "**extends**" para heredar ya que todo lo hará de manera interna.

FÓRMULA DEL MÉTODO: ***Object*****.toString(valor | nada)**

USO del MÉTODO → *suponiendo esta instancia=> Main clase= new Main();*

EJEMPLO VÁLIDO → Boolean.toString(true) = "true" ← Devuelve
EJEMPLO VÁLIDO → Float. toString(20.3f) = "20.3" ← Devuelve
EJEMPLO VÁLIDO → clase.toString() = "Main@2ff4acd0" ← Devuelve
EJEMPLO VÁLIDO → new String("Hola").toString(20.3f) = "Hola" ← Devuelve
EJEMPLO VÁLIDO → new Integer(1).toString(1) = "1" ← Devuelve

Argumentos: **1-opcional** → Espera recibir como argumento un valor proveniente de la clase usada, como ser Integer → espera recibir como argumento un valor numérico, un Boolean espera recibir como argumento un valor lógico, etc. Introduciremos un valor en nuestro argumento siempre y cuando sea la clase pura haciendo referencia al método, es decir, accedemos a este por medio de la clase en crudo, como se lo suele hacer para llamar a los miembros estáticos. **1-opcional** → Pero en el caso que estemos creando/instanciando una clase, ahí omitiremos el valor del argumento, ya que el valor que debemos introducir se lo haremos por medio del método constructor.

NOTA: lleva 1 argumento si accedemos al método estático por medio del nombre de la clase/tipo de dato | No llevará argumento si accedemos al método por medio de instancia/creación de objeto.
Este método se lo usará siempre y cuando nuestro **Object** no sea **null**, ya que de serlo nos lanzará una excepción de tipo **NullPointerException**. Este método es usado para saber el contenido de dicho objeto. Este método retornará el nombre de la clase que se ha instanciado y visualizará por consola la "**dirección de memoria**" donde se encuentra alojado.

CONSEJO: Podemos "**anular**" el método **toString()** desde nuestra clase creada para que nos dé más información útil y no el nombre de nuestra clase con un **código Hash**, la anulación lo haremos por medio de la anotación **@Override**, por lo que podremos reescribir nuestro propio método **toString()** haciendo que arroje otra clase de información, hasta más útil (se recomienda mucho la implementación de la anotación **@Override** para que podamos hacerlo personalizable y podamos dar una información más concisa).

MÉTODO ESTÁTICO → valueOf()

Descripción

- Este método se encargará de devolvernos cualquier valor de cualquier tipo de objeto en Java a un formato de tipo String, en pocas palabras convierte a String cualquier valor; este método tiene varias sobrecargas, por lo que al usarlo podrá aceptar cualquiera de estos tipos de valores: **bool** o **Boolean**, **Char**, **Char[]**, **double** o **Double**, **float** o **Float**, **int** o **Integer**, **long** o **Long**, **Object**.
  Los valores "null" de los objetos los podemos trabajar con tranquilidad que no nos lanzará una excepción, este método nos devolverá null en cadena si no encuentra valor alguno, a diferencia de **toString()** que nos arroja una excepción de tipo **NullPointerException**.

FÓRMULA DEL MÉTODO: ***Object*.valueOf(valor)**

USO del MÉTODO → *suponiendo esta instancia=>* ***Main clase= new Main();***

EJEMPLO VÁLIDO → Integer.valueOf(11) = "11" ← Devuelve
EJEMPLO VÁLIDO → Boolean. valueOf(true) = "true" ← Devuelve
EJEMPLO VÁLIDO → Float.valueOf(20.3f) = "20.3" ← Devuelve

Argumentos: 1-opcional → Espera recibir como argumento un valor proveniente de la clase usada, como ser Integer → espera recibir como argumento un valor numérico, un Boolean espera recibir como argumento un valor lógico, etc. Introduciremos un valor en nuestro argumento siempre y cuando sea la clase pura haciendo referencia al método, es decir, accedemos a este por medio de la clase en crudo, como se lo suele hacer para llamar a los miembros estáticos.

NOTA: lleva 1 argumento si accedemos al método estático por medio del nombre de la clase/tipo de dato.

Object.toString() vs String.valueOf() → Cuál deberíamos utilizar?

Ambos métodos tienen diferentes casos de usos, por lo que deberíamos utilizar ambos. Cuando creamos una nueva clase, debemos anular el método **toString()** con la anotación **@Override** para que podamos generar información útil sobre la instancia de nuestra clase, ya que por defecto si no lo hacemos nos mostrará el nombre de la clase más un **código Hash**, de tal forma → **Practica@4517d9a3**, por lo que dicha anotación nos dará la posibilidad de personalizar el tipo de mensaje a visualizar.
Cuando pasamos un objeto a **String.valueOf()**, llamará al método **toString()** en ese objeto y devolverá el resultado.
El método **toString()** nos permitirá personalizar la salida cuando queremos convertir un objeto en una cadena.
String.valueOf() nos permitirá convertir objetos en cadenas de forma más segura, sin necesidad de administrar valores nulos, como lo es en el caso de **Object.toString()**.
Cuando necesitamos convertir una instancia en una cadena, debemos usar el método **String.valueOf()** para garantizar la seguridad de nulos.

Más información:

- Repositorio del JDK (**importante**): https://github.com/openjdk/jdk
- https://openjdk.org/projects/jdk/
- https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/java/lang/String.html
- https://www.w3schools.com/java/java_ref_string.asp