MÓDULO PROFESIONAL PROGRAMACIÓN

UD 6 – Cadenas: la clase String

Java String is Special

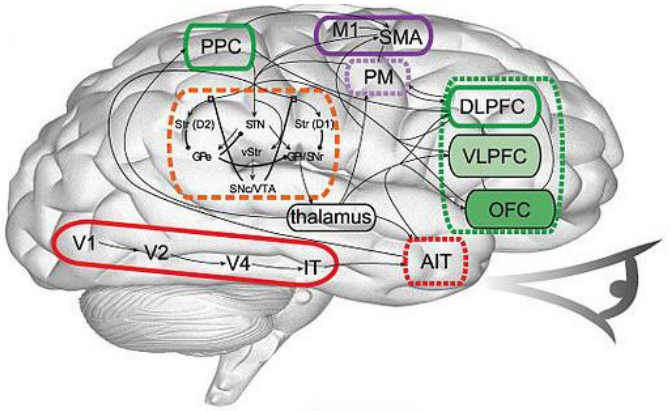
A recordar…

- **Estudia** la **teoría** asociada
  - Estudiar de la bibliografía recomendada (u otra de tu elección), solo con las diapositivas de clase no es suficiente.
  - Cuestiónate cosas, intenta encontrar la respuesta y su justificación.
- **Repasa** lo visto en **clase**
  - Realiza las actividades progresivamente
- **Ejercita** las **competencias**
  - Realiza los ejercicios progresivamente
  - Realiza todos los ejercicios que sean posibles (tanto de clase como otros)
- **Comprende** los **fundamentos** para obtener beneficios a largo plazo.

Pensamiento computacional

La esencia del pensamiento computacional es pensar como lo haría un científico informático cuando nos enfrentamos a un problema.

Pensamiento computacional



Características

- **Descomposición:** consiste en el procedimiento por el cual un problema de mayor complejidad se desarticula en pequeñas series más manejables.
- **Reconocimiento de patrones:** luego de la desarticulación del problema complejo, las pequeñas series son enfrentadas de forma individual de manera que puedan ser resueltas de forma similar a problemas frecuentados anteriormente.
- **Abstracción:** el proceso de abstracción, que consiste en decidir qué detalles debemos tener en cuenta y que podemos ignorar, es la base del pensamiento computacional.
- **Algoritmos:** se presentan pasos para la resolución de cada problema.

Cadenas de caracteres

Cadena de caracteres

Una cadena de caracteres es una secuencia ordenada de longitud arbitraria (aunque finita) de elementos que pertenecen a cierto alfabeto.

En general, una cadena de caracteres es una sucesión de caracteres (letras, números u otros signos o símbolos).

Cadena de caracteres

Desde un punto de vista de la programación, si no se ponen restricciones al alfabeto, una cadena podrá estar formada por cualquier combinación finita de todo el juego de caracteres disponibles.

(las letras de la ‘a’ a la ‘z’ y las de la ‘A’ a la ‘Z’, los números ‘0’ al ‘9’, el espacio en blanco ‘ ‘ y símbolos diversos ‘!’, ‘@’, ‘%’, etc.).

La clase String

En Java no existe un tipo de datos primitivo que sirva para la manipulación de cadenas de caracteres. En su lugar se utiliza una clase definida en la API de JAVA.

Esta clase es la clase String

La clase String es especial

Esto significa que en Java las **cadenas de caracteres** son, a todos los efectos, **objetos** que se manipulan como tales, sin embargo se diferencian de una clase común en:

- Que son **inmutables**, su contenido no puede ser modificado. Si quiero convertir una cadena a mayúsculas no se modifica el objeto existente, sino que se crea uno nuevo.
- Tienen un tratamiento especial en Java ya que su **uso es intensivo en programas**. Por tanto, la eficiencia en términos de computación y almacenamiento es crucial.

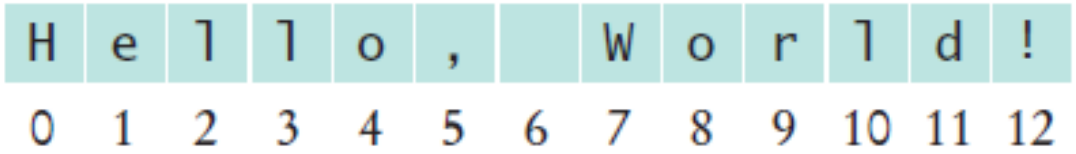
La clase String

La clase String forma parte del paquete java.lang (por tanto no es necesario importarlo, ya que este se importa automáticamente) y se describe completamente en la documentación del API del JDK:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

La clase String

Los caracteres de las cadenas tienen un índice que indica su posición. El primer carácter de una cadena tiene el índice 0, y el segundo el 1, el tercero el 2 y así sucesivamente. Para diferenciarse de los caracteres individuales los String se representan siempre con comillas dobles: “ejemplo”, “o”, “a”, “ ”, “12”



Declaración y creación

Las variables que almacenan direcciones de un objeto se denominan **variables de referencia**. Para definir una variable que se utilice como referencia a un objeto, se pone a su izquierda el nombre de la clase a la que pertenecerá el objeto, tal como:

```
String nombre;
```

Con esta sentencia, declaramos una variable que permite referenciar a un objeto de la clase String.

Declaración y creación

Si queremos crear una **cadena con un valor inicial**, podemos indicarlo de la siguiente forma:

```
String cadena = "valor inicial";                    //Construcción implícita
String cadena = new String("valor inicial");        //Construcción explícita
```

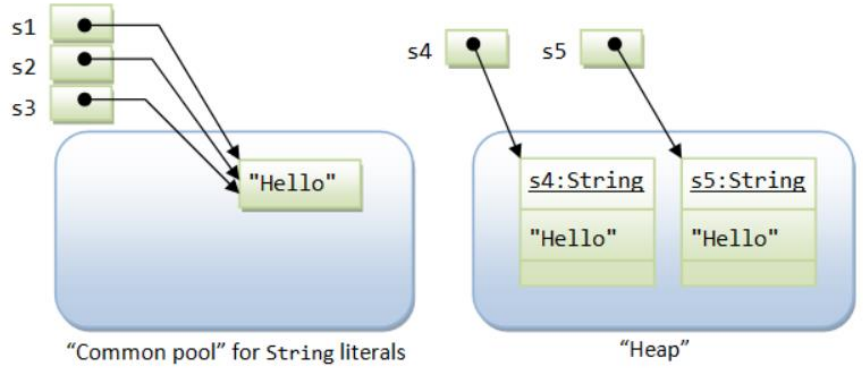
Para crear una **cadena vacía**

```
String cadena="";
String cadena=new String();
```

Declaración y creación

Como se ha mencionado, hay dos formas de construir una cadena: construcción **implícita asignando un literal de cadena** o creando **explícitamente un objeto String mediante el operador new** y el constructor. Por ejemplo,

```
String s1 = "Hello";                // literal String
String s2 = "Hello";                // literal String
String s3 = s1;                     // la misma referencia
String s4 = new String("Hello");    // objeto String
String s5 = new String("Hello");    // objeto String
```



Los literales con el mismo contenido, compartirán la misma ubicación de almacenamiento en el pool común. Mientras tanto, el objeto String almacenado en el heap, y no comparten ubicaciones de almacenamiento incluyendo dos objetos String tiene el mismo contenido.

Declaración y creación

¿Qué diferencia existe entre las tres **a** de este código?

```
int a=10;
char c='a';
String cadena= "a";
```

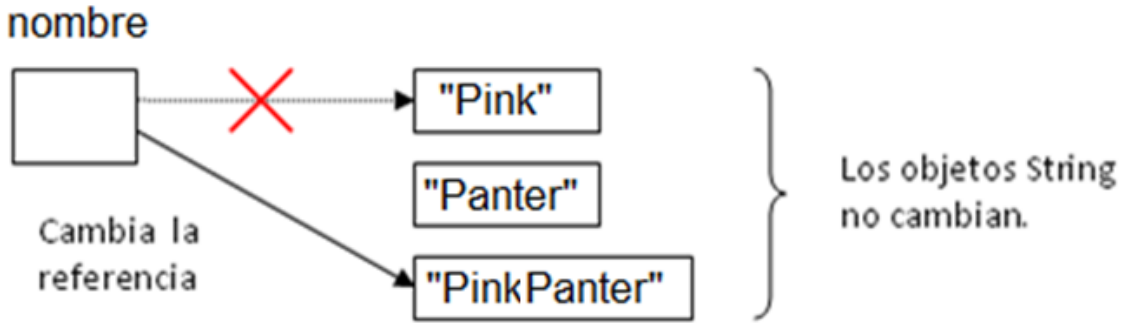
Un caso especial de cadena es la que contiene o caracteres, a esta cadena se la llama cadena vacía y se representa con comillas dobles (""), es el valor por defecto de cualquier String.

¿Qué diferencia hay entre "" y " " ?

Declaración y creación

Los Strings no se modifican una vez que se les ha asignado un valor. Es decir, son **inmutables**. Si se produce una reasignación se crea un nuevo objeto String con el nuevo contenido y el anterior será borrado.

```
String nombre;
nombre = "Pink";
nombre = nombre + "Panter";
```



Declaración y creación

Un objeto se considera inmutable si su estado no puede cambiar después de su construcción.

Lectura y escritura

Para mostrar el contenido de un String por pantalla se hace igual que si se tratara de un tipo primitivo de Java:

```
String nombre="María";
System.out.println("Tu nombre es: "+ nombre);
```

Para poder realizar la lectura de cadenas, en la clase Scanner también existe un método nextLine() o next() que permite realizar su lectura como con el resto de tipos primitivos:

```
Scanner sc=new Scanner(System.in);
System.out.println("¿Cómo te llamas?");
String nombre= sc.nextLine();
```

Lectura y escritura

Limpiar el buffer de entrada

Al introducir un dato numérico de cualquier tipo, se introduce dicho número más el carácter de intro ('\n'), al realizar la lectura, solamente se lee el número y el salto de línea permanece en el buffer de lectura.

Esto puede ser problemático si después de leer un número, se lee una cadena de caracteres, ya que esta tomará ese ultimo salto de línea del teclado en lugar de realizar una lectura. Para evitar este problema, basta con realizar una lectura extra para eliminar dicho salto de línea.

Lectura y escritura

```java
import java.util.Scanner;
public class EjemploLimpiarBufferEntrada {
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Dime un número");
        int num=sc.nextInt();           // lectura del número

        System.out.println("Dime un nombre");

        sc.nextLine();                  // eliminamos lo que pueda haber en el buffer de lectura

        String cadena=sc.nextLine();    // lectura de la cadena
        System.out.println("has introducido el número "+num+" y el nombre "+cadena);
    }
}
```

Operaciones con String

La clase String implementa una gran cantidad de operaciones muy útiles cuando se trabaja con cadenas:

- **Concatenar** cadenas.
- **Comparar** cadenas
- Obtener el **tamaño** de una cadena
- **Eliminar espacios en blanco**
- Devolver el carácter que ocupa la **posición i-esima**
- Devolver una **subcadena**
- Devolver la **posición** que ocupa un **carácter de la cadena.**
- Devolver la **posición** que ocupa una **subcadena en la cadena.**
- Transformar la cadena en **mayúsculas o minúsculas.**
- **Reemplazar un carácter por otro.**

Operaciones con String

Recuerda que siempre que se utilices un **método perteneciente a un objeto**, se pondrá el nombre del objeto seguido del nombre del método separados por un punto:

```
unObjeto.metodo();
```

Siempre se terminará con dos paréntesis, tenga o no parámetros. Los parámetros son datos que se envían al método para utilizarlos en su ejecución.

CONCATENACIÓN

Una de las principales operaciones con Strings es la concatenación, para esto tenemos 2 métodos el operador + y el método concat(String):

```
String cad ="Hola";
cad = cad +" mundo";
cad = cad.concat(" como estás");
System.out.println(cad);
```

La salida de este código muestra por pantalla el mensaje "Hola mundo como estás" ambos métodos concatenan cadenas de la misma forma, pero el método **concat() es 10 veces más eficiente,** por lo que se recomienda su uso en concatenaciones sucesivas.

COMPARACIÓN

Para comparar cadenas y ver si 2 cadenas son iguales utilizamos el método equals()

```
String cad1="hola";
String cad2="HOLA";
if(cad1.equals(cad2))
    System.out.println("son iguales");
```

El método **boolean equals(String)** devolverá verdadero si ambas son iguales, si no lo son devolverá falso. Si queremos que esta comparación se haga ignorando mayúsculas y minúsculas utilizaremos:

boolean equalsIgnoreCase(String)

CARÁCTER EN POSICIÓN

Para conocer el carácter que hay almacenado en una determinada posición del String disponemos del siguiente método: char charAt(int)

```
String cad1="hola";
System.out.println(cad1.charAt(3)); //esta salida mostrará 'a'
```

¿Por qué devuelve 'a' y no 'l'?

Si al utilizar charAt(int) intentamos acceder a una posición del String que no existe, se producirá un error que abortará la ejecución del programa.

LONGITUD

Otra de las operaciones básicas con cadenas de caracteres es obtener su tamaño, para esto utilizaremos: int length()

```
String cad="hola ";
System.out.println(cad.length());   //esta salida mostrará 5
```

TRIM

Para casos como el anterior puede ser útil eliminar los espacios en blanco al inicio y/o al final de una cadena. Para esto utilizaremos: String trim()

```
String cad="hola ";
cad=cad.trim();
System.out.println(cad.length()); //esta salida mostrará 4
```

COMPARACIÓN ALFABÉTICA

También es útil poder comparar 2 cadenas para ordenarlas alfabéticamente. Para esto podemos utilizar el método: int compareTo(String)

```
String cad1="hola";
String cad2="HOLA";
System.out.println(cad1.compareTo(cad2));
```

Este método devuelve

- un valor negativo si la primera cadena precede alfabéticamente a la segunda
- o si son iguales
- o un número positivo si es la segunda la que precede.

Si queremos ignorar las mayúsculas utilizaremos: int compareToIgnoreCase(String)

SUBCADENAS

Además de buscar un carácter, también se puede buscar una subcadena de la siguiente forma: String substring(int[, int])

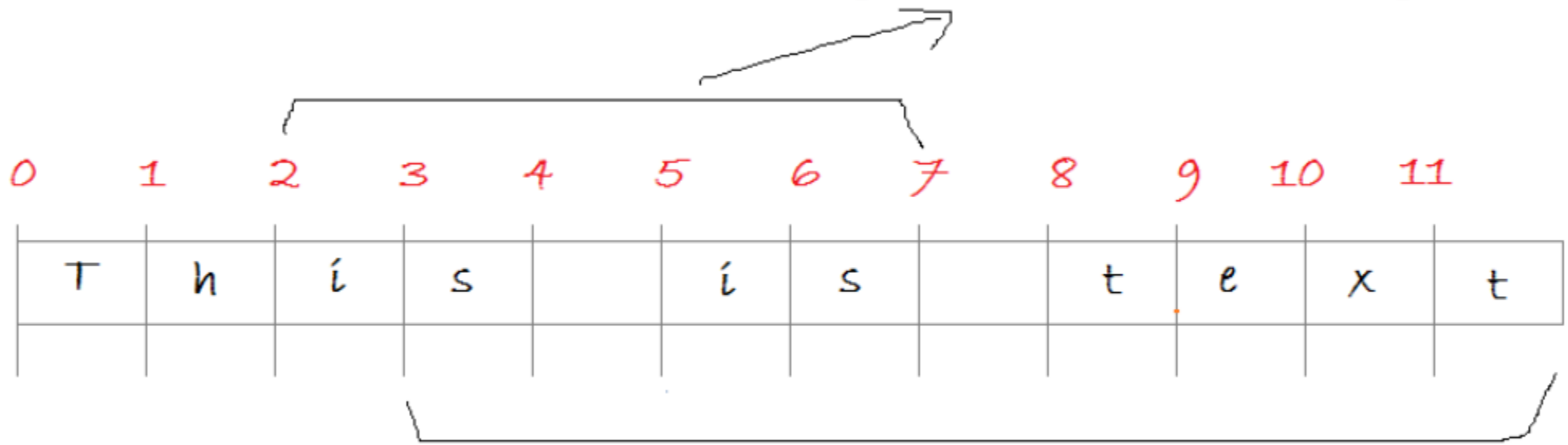
```
String cad1="hola mundo";
System.out.println(cad1.substring(0,4)); //esta salida mostrará "hola"
System.out.println(cad1.substring(5));  //esta salida mostrará "mundo"
```

Podemos recuperar una **subcadena indicando 1º valor**: desde esa posición hasta el final. Si **indicamos 2 valores** estos son la posición de inicio (inclusivo) y la de fin(exclusivo).

Al igual que al utilizar charAt(int) intentamos acceder a una posición del String que no existe, se producirá un error que abortará la ejecución del programa.

String str = "This is text";

String substr = str.substring(2,7)



String substr = str.substring(3)

BÚSQUEDA

De igual forma podemos buscar cual es la primera ocurrencia de un carácter o una cadena:

int indexOf(char)

int indexOf(String)

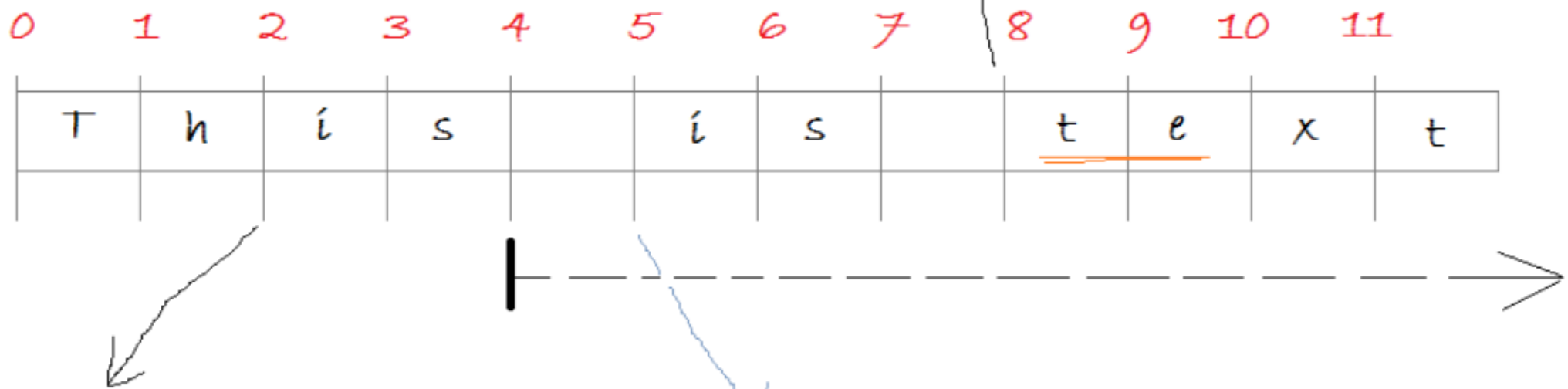
```
String cad1="hola mundo";
System.out.println(cad1.indexOf('o'));      //muestra 1
System.out.println(cad1.indexOf("mundo"));  //muestra 5
System.out.println(cad1.indexOf("Pepe"));   //muestra -1
```

Si el carácter o cadena introducidos no se encuentran se devolverá un -1

Si hay más de una ocurrencia se devuelve la primera. Si quisiéramos la última ocurrencia utilizaríamos: int lastIndexOf(String) o lastIndexOf(Char)

```
String str = "This is text";
```

```
int idx = str.indexOf("te");
```




```
int idx = str.indexOf('i');
```

```
idx = str.indexOf('i', 4)
```

MAYÚSCULAS Y MINÚSCULAS

Aunque algunos métodos tienen sus variantes para ignorar las mayúsculas minúsculas, lo más conveniente es transformarlas explícitamente antes de trabajar con ellas: String toUpperCase(), String toLowerCase()

```
String cad1="Hola Mundo";
System.out.println(cad1.toUpperCase()); //muestra "HOLA MUNDO"
System.out.println(cad1.toLowerCase()); //muestra "hola mundo"
```

REEMPLAZO

En Java no sólo podemos buscar un carácter o cadena, también se puede remplazar con los métodos:

String replace(char, char),

String replaceAll(String, String),

String replaceFirst(String, String)

```
String cad1="Hola Mundo";
System.out.println(cad1.replace('o','e'));          //muestra "Hela Munde"
System.out.println(cad1.replaceAll ("Mu", "a"));    //muestra "Hola ando"
System.out.println(cad1.replaceFirst("o", "i"));    //muestra "Hila Mundo"
```

Si el carácter o cadena introducidos no se encuentran se devolverá la cadena original

FORMATO

Una de las formas para dar formato a una cadena en Java es utilizando el método **estático** **String.format().** Su uso es el siguiente:

String.format(**cadena-de-formato**, arg1, arg2, arg3, …);

La **cadena de formato** contiene texto normal y los denominados especificadores de formato.

```
String salida = String.format("%s = %d", "Ana", 35);
System.out.println(salida);
```

Salida

```
Ana = 35
```

FORMATO

A continuación se ofrece una referencia rápida de todos los **especificadores de formato** admitidos:

conversion-character	Description	Type
d	decimal integer	byte, short, int, long
f	floating-point number	float, double
b	Boolean	Object
B	will uppercase the boolean	Object
c	Character Capital	String
C	will uppercase the letter	String
s	String Capital	String
S	will uppercase all the letters in the string	String
h	hashcode - A hashcode is like an address. This is useful for printing a reference	Object
n	newline - Platform specific newline character - use **%n** instead of **\n** for greater compatibility	

FORMATO: ejemplos

Formato a enteros

Con el especificador de formato %d, puede utilizar un argumento de todos los tipos enteros , incluyendo byte, short, int y long.

```
String salida = String.format("%d", 93); // devuelve a 93
```

Especificando un ancho

```
String salida = String.format("|%20d|", 93); // devuelve |                  93|

//justificado a la izquierda dentro de un ancho especificado
String salida = String.format("|%-20d|", 93); // devuleve |93                  |
```

FORMATO: ejemplos

Rellenando con ceros

```
String salida = String.format("|%020d|", 93); // devuelve |00000000000000000093|
```

Más ejemplos

```
String nombre = "David";
int hora=6;
int minutos=5;

String salida = String.format("%s ha accedido a las %02d:%02d h",nombre,hora,minutos);
//Devuelve David ha accedido a las 06:05 h
```

Formato de decimales

```
String salida = String.format("%.2f", 8.2476);  //devuelve 8.25
```

```
String salida = String.format("%.2f", 8.2476);  //devuelve 8.25
```

FORMATO

String.format(**cadena-de-formato**, arg1, arg2, arg3, …);

La **cadena de formato** contiene texto normal y los especificadores de formato. El texto normal (incluidos los espacios en blanco) se imprimirá tal como está. Los especificadores de formato tienen la forma de:

%[flags][width][.precision]conversion-character

- [flags] (banderas) opcionales y se usan para controlar la alineación, el relleno y otros.
- [width] (ancho) es opcional y se usa para especificar el ancho de la salida.
- [precision] es opcional, se suele utilizar para especificar el número de dígitos decimales para valores reales.

Más operaciones con String

Además de los métodos comentados, en Java existen muchos más para realizar otras tareas:

- Transformar de entero, decimal etc. a String
- Transformar de String a entero, decimal, etc..
- Partir una cadena
- Y muchos mas…

Clases envolventes para tipos primitivos

Cada tipo primitivo en Java tiene su clase envolvente (wrapper class), cuyos objetos pueden almacenar un valor numérico del tipo de la variable primitiva.

- Estos objetos se suelen utilizar para aprovechar los métodos asociados a estos objetos que permiten entre otras cosas la conversión de datos de unos tipos primitivos a otros o desde una cadena a un número y viceversa.
- Las clases envolventes son **Integer**, Byte, Short, **Boolean**, **Character**, **Long**, **Float** y **Double**. Y como se puede observar tienen nombres similares a los tipos primitivos y están escritos con la primera letra en mayúscula.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Clases envolventes

Esta solución elegida se conoce técnicamente también con el nombre “decorador”

Representación gráfica que simboliza el concepto "decorador"

Consiste en envolver dos o más objetos, uno dentro del otro, como si se trataran de muñecas rusas.

Cada envoltorio aporta una cierta funcionalidad extra ("decora" el objeto interno de forma diferente al original). De esta manera es posible crear instancias "a gusto del consumidor", seleccionando sólo la funcionalidad que sea necesaria.

Clases envolventes para tipos primitivos

A continuación, se recogen ejemplos de sentencias de creación de objetos de las clases envolventes:

```
Integer entero = new Integer (45);
Boolean escierto = new Boolean(false);
Character letra= new Character('b');
Long enterolargo = new Long (1234);
Float realsimple = new Float(1234);
Double realDoble = new Double(1234);
```

Actividad. Pasar de cadena a número y viceversa

Las clases envolventes nos permitirán, a través de métodos, realizar operaciones de conversión como se puede ver en los siguientes ejemplos que convierten cadenas a tipos primitivos y de tipos primitivos a cadena.

```java
public class PruebaToNumber {

    public static void main(String[] args) {

        String numString="123";
        System.out.println("String: "+numString);

        long numLong=Long.parseLong(numString);
        System.out.println("long: "+numLong);

        int numInt=Integer.parseInt(numString);
        System.out.println("int: "+numInt);

        short numShort=Short.parseShort(numString);
        System.out.println("short: "+numShort);

        byte numByte=Byte.parseByte(numString);
        System.out.println("byte: "+numByte);

        double numDouble=Double.parseDouble(numString);
        System.out.println("double: "+numDouble);

        float numFloat=Float.parseFloat(numString);
        System.out.println("Float: "+numFloat);

    }
}
```

```java
public class PruebaToString{

    public static void main(String[] args) {

        long numLong=1;
        String cad1=Long.toString(numLong);

        int numInt=2;
        String cad2=Integer.toString(numInt);

        short numShort=3;
        String cad3=Short.toString(numShort);

        byte numByte=4;
        String cad4=Byte.toString(numByte);

        double numDouble=5;
        String cad5=Double.toString(numDouble);

        float numFloat=6;
        String cad6=Float.toString(numFloat);

        System.out.println(cad1+cad2+cad3+cad4+cad5+cad6);

    }
}
```

Clases envolventes para tipos primitivos

char a String: String cadena = Character.toString(char)

Ejemplo,

```
char codigo ='A'
String cadena = Character.toString(codigo)
```

String a char: char caracter = cadena.charAt(o);

Ejemplo,

```
String codigo = "Hola";
char caracter = codigo.chatAt(0);
```

Clases envolventes para tipos primitivos

String a boolean: boolean flag= Boolean.parseBoolean(cadena)

Ejemplo,

```
boolean flag= Boolean.parseBoolean("true")
```

boolean a String String cadena= Boolean.toString(flag);

Ejemplo,

```
boolean flag = true;
String cadena= Boolean.toString(flag);
```

ValueOf

El método **valueOf** es un método sobrecargado aplicable a numerosas clases de **Java** y que permite realizar conversiones de tipos. Cuando se usa con la clase String, convierte diferentes tipos de valores en cadena.

```
public class StringValueOfExample{
    public static void main(String[] args) {

        int value=30;
        String s1=String.valueOf(value);
        System.out.println(s1+10);  //concatena la cadena con 10

    }
}
```

ValueOf con clases envolventes

El siguiente ejemplo muestra el uso de valueOf con clases envolventes de tipos primitivos

```
public class ValorDe
{
    public static void main(String []args)
    {
        int x = 10;
        Integer a = Integer.valueOf(x);
        Integer b = Integer.valueOf("1024");
        Double d = Double.valueOf(b);
        Float e = Float.valueOf(a);

        System.out.println(a);
        System.out.println(b);
        System.out.println(d);
        System.out.println(e);
    }
}
```

No todas las conversiones son posibles.

NO SE IMPARTE

Cadenas MUTABLES

StringBuilder y StringBuffer

Cadenas mutables

Lo primero que habríamos de preguntarnos es la razón por la que existen múltiples tipos de datos en Java para operar sobre una misma categoría de información: las **cadenas de caracteres.**

Dado que el tipo String es inmutable (no podemos modificar su contenido), cualquier operación de modificación sobre una variable de este tipo, implica la creación de un nuevo objeto String.

Cadenas mutables

Obviamente, el hecho de se liberen y creen nuevos objetos String cada vez que se cambia su contenido influye también en el rendimiento de los programas. El recolector de basura de Java tendrá más trabajo.

No obstante, la decisión de hacer inmutable el tipo String parte de análisis realizados sobre aplicaciones en los que se observa que en una gran proporción de los casos su contenido no es modificado, por lo que los beneficios obtenidos son, en general, superiores a los inconvenientes.

Cadenas mutables

Dado que en una aplicación puede surgir la **necesidad de alterar de manera frecuente el contenido de una cadena de caracteres**, Java nos ofrece tipos de datos específicos para operar sobre ellas.

Las **cadenas de caracteres mutables**, representadas por los tipos **StringBuilder** y **StringBuffer**, se comportan como cualquier otro tipo de dato por referencia en Java: se asigna memoria dinámicamente según es necesario.

Cadenas mutables

Dado el siguiente ejemplo:

```
StringBuilder saludo = new StringBuilder("Hola");
saludo.append(" mundo");
```

Al agregar al final la segunda cadena a la primera, sencillamente se actualiza el contenido inicial de la variable saludo, de tipo **StringBuilder**, en lugar de liberarse el objeto original y crearse otro nuevo.

En general, un programa que vaya a modificar con cierta frecuencia el contenido de una o más cadenas de caracteres obtendrá mejor rendimiento de esta forma que con el tipo String original.

Cadenas mutables

Habitualmente cada ventaja conlleva algún tipo de inconveniente. La flexibilidad de los tipos StringBuilder y StringBuffer también tiene su contrapartida.

Al contar con un contenido mutable, **StringBuilder no es un tipo de dato seguro para aplicaciones con múltiples hilos de ejecución.** Si dos hilos acceden simultáneamente para cambiar algo en la cadena, el resultado puede ser totalmente inesperado.

Cadenas mutables

¿Qué hacer si necesitamos trabajar con cadenas de caracteres mutables en un entorno multi-hilo? Usar el tipo **StringBuffer** en lugar de StringBuilder.

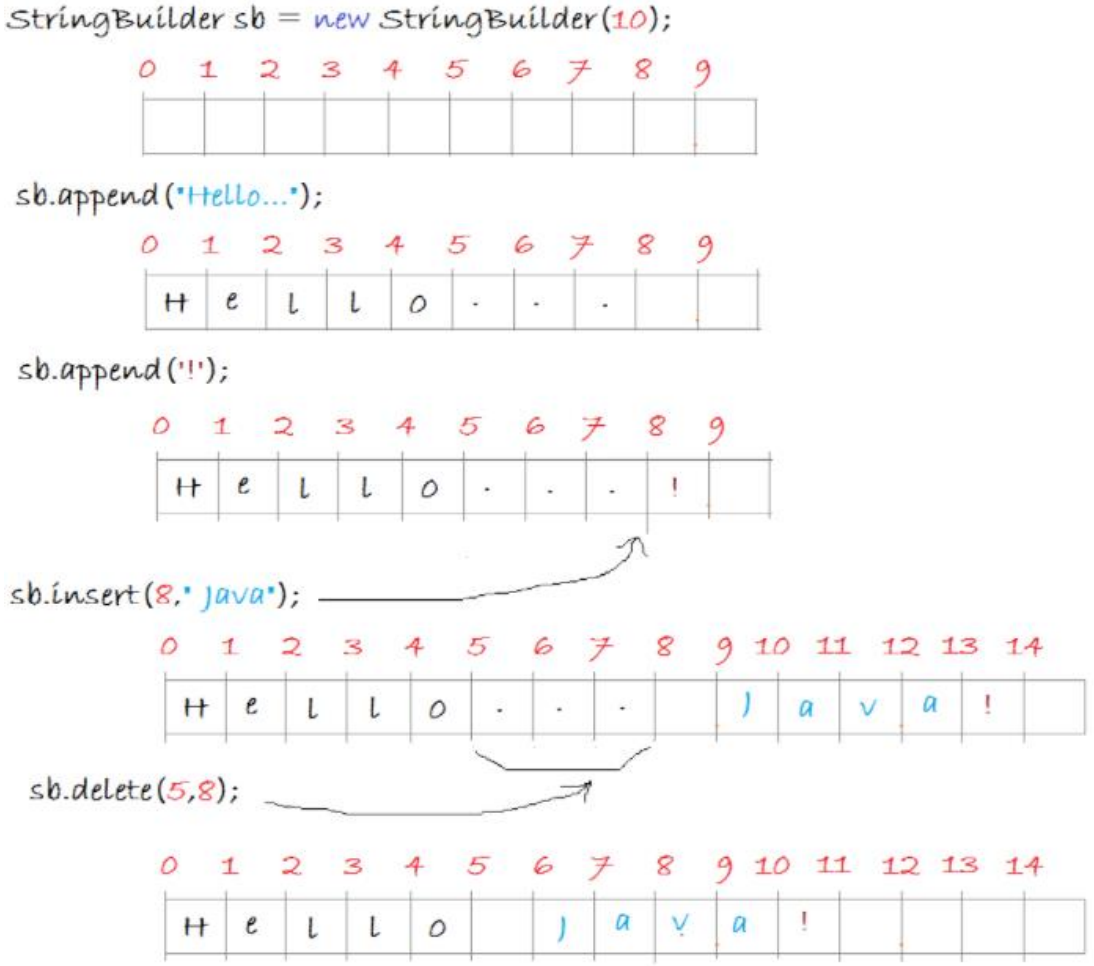
Ambos son prácticamente idénticos en cuanto a funcionalidad se refiere, pero internamente la implementación de todos los métodos que alteran la cadena está sincronizada en el caso de StringBuffer.

Es decir, **este último tipo es seguro (thread-safe) para múltiples hilos**, mientras que StringBuilder no lo es. Esta seguridad se obtiene a costa del rendimiento, ya que la sincronización provoca que las operaciones sobre cadenas con StringBuffer sean más lentas que con StringBuilder o que con String.

StringBuilder y StringBuffer

Como vamos a trabajar con programas de un solo hilo (el del main), veremos el uso de **StringBuilder** como cadenas mutables

La clase StringBuilder




```java
public class StringBuilderDemo {

    public static void main(String[] args) {

        // Se creo un objeto StringBuilder
        // sin caracteres y con una capacidad inicial indicada
        // en el argumento del constructir
        StringBuilder sb = new StringBuilder(10);

        // Concateno al final de sb Hello ...
        sb.append("Hello...");
        System.out.println("- sb luego de concatenar una cadena: " + sb);

        // Concateno un caracter
        char c = '!';
        sb.append(c);
        System.out.println("- sb luego de concatenar un caracter: " + sb);

        // Inserto ina cadena en el índice 8
        sb.insert(8, " Java");
        System.out.println("- sb luego de insertar una cadena: " + sb);


        // Borro una subcadena del índice 5 al 7 inclusive
        sb.delete(5,8);

        System.out.println("- sb luego de borrar: " + sb);
    }
}
```

String	StringBuilder	StringBuffer
Immutable	Mutable	Mutable
Thread Safe	No Thread Safe	Thread Safe
Use String, when it's not going to change much	Use StringBuilder, when it's going to change often with Single Thread	Use StringBuffer, when it's going to change often with Many Thread Involved

Recursos

- Curso youtube : Java desde 0
- Libro Java 9. Manual imprescindible. F. Javier Moldes Teo. Editorial Anaya
- App SoloLearn: Aprende a Programar. Curso Java