

**MP0485**  
**Programación**  
**UF4. Estructuras de datos compuestos**

## **4.2. Cadenas de caracteres**

# Índice

---

☰	Objetivos	3
☰	String	4
☰	Resumen de métodos de la clase String	12
☰	StringTokenizer	13
☰	StringBuilder	15
☰	StringBuffer	18
☰	Resumen	19

# Objetivos

---

Con esta unidad perseguimos los siguientes objetivos:

1

Distinguir las clases Java más importantes para el manejo de cadenas de caracteres (*String, StringBuffer, StringBuilder, StringTokenizer*).

2

Utilizar los atributos y métodos más importantes asociados a los objetos que representan cadenas de caracteres.

---

¡Ánimo y adelante!

# String

Recuerda que Java no tiene un tipo de dato elemental para representar cadenas de caracteres. Para almacenar un texto debemos crear un objeto de una de las clases disponibles para el manejo de datos alfanúmericos o cadenas.

Recuerda que siempre que queremos construir un objeto a partir de una clase debemos utilizar el operador `new`. Veamos un pequeño programa que declara una referencia a un objeto de la clase `String`, construye el objeto y lo muestra en pantalla:

```
public class Principal {  
    public static void main(String[] args) {  
        String texto;  
        texto = new String("La cripta mágica");  
        System.out.println(texto);  
    }  
}
```

La variable `texto` es una referencia a un objeto, contiene la dirección de memoria RAM donde se encuentra el nuevo objeto creado. Podemos declarar la referencia al objeto y asignarle valor en la misma línea:

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = new String("La cripta mágica");  
        System.out.println(texto);  
    }  
}
```

Recuerda que, aunque los objetos se construyen con el operador `new`, en el caso de los objetos de la clase `String`, como excepción, podemos omitir dicho operador y simplificar la sentencia de la siguiente manera:

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "La cripta mágica";  
        System.out.println(texto);  
    }  
}
```



De manera implícita se está aplicando igualmente el operador `new`.

Sabemos que `texto` no contiene un dato elemental, sino un objeto más complejo compuesto por atributos o propiedades y métodos o funciones.

¡Vamos a descubrir los métodos más importantes de la clase `String`!

1

### El método `length()`

El método `length()` devuelve la longitud de la cadena, es decir, el número de caracteres que contiene.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "La cripta mágica";  
        System.out.println(texto);  
        System.out.println("Longitud: "+texto.length());  
    }  
}
```

2

### El método `toUpperCase()`

El método `toUpperCase()` convierte a mayúsculas la cadena a la que se aplica.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "La cripta mágica";  
        System.out.println(texto);  
        System.out.println(texto.toUpperCase());  
    }  
}
```

3

**Método `toLowerCase()`**

El método `toLowerCase()` convierte a minúsculas la cadena a la que se aplica.

```
System.out.println(texto.toLowerCase());
```

4

**Comparando cadenas: `equals()` y `equalsIgnoreCase()`**

El método `equals()` compara la cadena del objeto al que se aplica con otra cadena pasada como argumento. Si las dos cadenas son iguales devuelve `true` y si las dos cadenas son distintas devuelve `false`.

El método `equalsIgnoreCase()` funciona igual que el método `equals()` con la diferencia de que no distingue entre mayúsculas y minúsculas. En el ejemplo que sigue la primera comparación es `false` (las cadenas son distintas) y la segunda es `true` (las dos cadenas son iguales porque no se distingue entre mayúsculas y minúsculas).

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "La cripta mágica";  
        System.out.println(texto);  
        boolean iguales1 = texto.equals("LA CRIPTA MÁGICA");  
        boolean iguales2 = texto.equalsIgnoreCase("LA CRIPTA MÁGICA");  
        System.out.println(iguales1); // Muestra false, no son iguales  
        System.out.println(iguales2); // Muestra true, son iguales  
    }  
}
```

No debes comparar cadenas de caracteres utilizando el operador `==`, puedes llevarte sorpresas inesperadas. La forma correcta es utilizar el método `equals()` que acabas de aprender. Más adelante comprenderás el motivo cuando entiendas mejor el funcionamiento de los objetos.

5

**Métodos `compareTo()` y `compareToIgnoreCase()`**

El método `compareTo()` compara la cadena del objeto a la que se aplica con la cadena pasada como argumento. Si son iguales devuelve 0, si la primera cadena es menor que la segunda devuelve un número negativo, y si la primera cadena es mayor que la segunda devuelve un número positivo. El método `compareToIgnoreCase()` funciona igual pero no distingue entre mayúsculas y minúsculas.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Pepe";  
        System.out.println(texto.compareTo("Pepe")); // Muestra 0 ya que "Pepe" = "Pepe"  
        System.out.println(texto.compareTo("Rosa")); // Muestra -2 ya que "Pepe" < "Rosa"  
        System.out.println(texto.compareTo("Alicia")); // Muestra 15 ya que "Pepe" > "Alicia"  
    }  
}
```

6

### Método *charAt()*

El método *charAt()* devuelve el carácter que ocupa la posición indicada en el argumento. El primer carácter ocupa la posición 0.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        System.out.println(texto.charAt(0)); // Muestra C  
        System.out.println(texto.charAt(3)); // Muestra o  
    }  
}
```

Con ayuda de una estructura *for* y el método *length()* podemos recorrer todos los elementos de la cadena.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        for (int i=0; i<texto.length(); i++) {  
            System.out.println(texto.charAt(i));  
        }  
    }  
}
```

Y si queremos escribir en pantalla los caracteres al revés, podemos hacerlo así:

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        for (int i=texto.length()-1; i>=0; i--) {  
            System.out.print(texto.charAt(i));  
        }  
    }  
}
```

Observa que estamos ejecutando el método *print()* en lugar de *println()*. La diferencia es que *println()* genera un salto de línea al final de la cadena mostrada y *print()* no. De esta forma podemos escribir la cadena "Cocodrilo" al revés en una sola línea.

Recuerda también que internamente cada carácter está representado como un número, y basta hacer una conversión o *cast* a tipo *int* para mostrar el código ASCII asociado a cada uno de los caracteres de la cadena.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        for (int i=0; i<texto.length(); i++) {  
            System.out.println((int)texto.charAt(i));  
        }  
    }  
}
```

Puedes obtener el mismo resultado utilizando el método *codePointAt()* que veremos a continuación.

7

#### El método *codePointAt()*

El método *codePointAt()* devuelve el código ASCII asociado al carácter de la posición pasada como argumento.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        for (int i=0; i<texto.length(); i++) {  
            System.out.println(texto.codePointAt(i));  
        }  
    }  
}
```

8

#### El método *concat()*

El método *concat()* devuelve una nueva cadena, que es el resultado de concatenar la cadena del objeto al que se aplica con la cadena pasada como argumento.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo ";  
        String texto2 = texto.concat("Dundee");  
        System.out.println(texto2); // Muestra Cocodrilo Dundee  
    }  
}
```

9

### El método `contains()`

El método `contains()` devuelve `true` si la cadena del objeto al que se aplica contiene la subcadena especificada en el argumento.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        System.out.println(texto.contains("dri")); // Muestra true  
        System.out.println(texto.contains("pot")); // Muestra false  
    }  
}
```

10

### Los métodos `startsWith()` y `endsWith()`

Los métodos `startsWith()` y `endsWith()` comprueban si la cadena del objeto al que se aplican empieza o termina con la subcadena especificada como argumento.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        System.out.println(texto.startsWith("Coc")); // Muestra true  
        System.out.println(texto.endsWith("lo")); // Muestra true  
    }  
}
```

11

### El método `indexOf()`

El método `indexOf()` devuelve la posición que ocupa la primera ocurrencia del carácter especificado como argumento dentro de la cadena del objeto al que se aplica. Si el carácter no se encuentra dentro de la cadena devuelve -1.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        int posicion = texto.indexOf('d');  
        System.out.println(posicion); // Muestra 4  
    }  
}
```

También puede utilizarse para localizar la posición de una subcadena en lugar de un solo carácter.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Cocodrilo";  
        int posicion = texto.indexOf("ril"); // Muestra 5  
        System.out.println(posicion);  
    }  
}
```

12

### Los métodos *replace()*, *replaceAll()* y *replaceFirst()*

Los métodos *replace()*, *replaceAll()* y *replaceFirst()* sirven para reemplazar parte de la cadena del objeto al que se aplican, aunque funcionan de manera diferente.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Este gato es un gato persa que se come la comida de mi gato";  
        String cadena1 = texto.replace('a', 'e');  
        String cadena2 = texto.replaceFirst("gato", "perro");  
        String cadena3 = texto.replaceAll("gato", "perro");  
        System.out.println(cadena1); // Muestra "Este geto es un geto perse que se come le comide de mi ge  
        System.out.println(cadena2); // Muestra "Este perro es un gato persa que se come la comida de mi g  
        System.out.println(cadena3); // Muestra "Este perro es un perro persa que se come la comida de mi  
  
    }  
}
```

La expresión *texto.replace('a', 'e')* devuelve una nueva cadena que resulta de sustituir en *texto* todos los caracteres 'a' por 'e'. Puede aplicarse también a subcademas, con lo que funcionaría igual que *replaceAll()*.

La expresión *texto.replaceFirst("gato", "perro")* devuelve una nueva cadena que resulta de sustituir en *texto* la primera ocurrencia de la subcadena "gato" por la subcadena "perro".

La expresión *texto.replaceAll("gato", "perro")* devuelve una nueva cadena que resulta de sustituir todas las ocurrencias de la subcadena "gato" por la subcadena "perro".

13

### La función *split()*

La función *split()* devuelve un *array* de objetos *String* que resulta de la división de la cadena del objeto al que se aplica en subcademas divididas por el carácter especificado como argumento.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "Hola-Hello-Hallo-Ciao-Ni hao-salut-ahoj";  
        String[] subcademas = texto.split("-");  
        for (int i=0; i<subcademas.length; i++) {  
            System.out.println(subcademas[i]);  
        }  
    }  
}
```

El programa anterior divide la cadena *texto* en subcadenas divididas por el carácter "-". El resultado queda guardado en el array de objetos *String* llamado *subcadenas*, que luego recorremos por medio de una estructura *for*.

El resultado de la ejecución del programa es:

```
Hola  
Hello  
Hallo  
Ciao  
Ni hao  
salut  
ahoj
```

14

#### El método *trim()*

El método *trim()* devuelve una nueva cadena que resulta de recortar en la cadena original los espacios en blanco que haya al principio y al final.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = " La crita mágica ";  
  
        System.out.println("---"+texto+"--");  
        System.out.println("---"+texto.trim()+"--");  
    }  
}
```

En el ejemplo estamos mostrando primero la cadena tal cual y luego la cadena recortando los espacios. Concatenamos con dos guiones al principio y al final para que se aprecien los espacios en blanco.

15

#### El método *substring()*

El método *substring()* extrae de la cadena principal una subcadena. Puede utilizarse de dos formas distintas. Si pasamos un número entero como argumento, devuelve una subcadena a partir de la posición indicada en el argumento y hasta el final. Si pasamos dos números enteros como argumentos, estos son interpretados como las posiciones de inicio y fin de la subcadena a extraer.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto = "La crita mágica";  
  
        String cadena1 = texto.substring(3);  
  
        String cadena2 = texto.substring(3, 8);  
        System.out.println(cadena1); // Escribe "cripta mágica".  
        System.out.println(cadena2); // Escribe "cripta".  
    }  
}
```

# Resumen de métodos de la clase String

En esta apartado puedes ver un breve resumen de los métodos que hemos visto de la clase *String*.

- El método **length()**: devuelve el número de caracteres de la cadena.
- El método **toUpperCase()**: devuelve la cadena convertida a mayúsculas.
- Método **toLowerCase()**: devuelve la cadena convertida a minúsculas.
- Método **equals(argumento)**: compara la cadena con otra cadena pasada como argumento y devuelve *true* si son iguales.
- Método **equalsIgnoreCase(argumento)**: funciona igual que *equals()* pero no distingue entre mayúsculas y minúsculas.
- Método **compareTo(argumento)**: compara una cadena con la cadena pasada como argumento. Si son iguales devuelve 0, si la primera es menor que la segunda devuelve un número negativo, y si es mayor un número positivo.
- Método **charAt(posición)**: devuelve el carácter que ocupa la posición especificada como argumento.
- El método **codePointAt()**: devuelve el código ASCII asociado al carácter de la posición pasada como argumento.
- El método **concat(argumento)**: devuelve una cadena concatenada con la cadena pasada como argumento.
- El método **contains(subcadena)**: devuelve *true* si la cadena contiene la subcadena pasada como argumento, de lo contrario devuelve *false*.
- El método **startsWith(subcadena)**: devuelve *true* si la cadena comienza por la subcadena especificada en el argumento, de lo contrario devuelve *false*.
- El método **endsWith(subcadena)**: devuelve *true* si la cadena termina por la subcadena especificada como argumento, de lo contrario devuelve *false*.
- El método **indexOf(carácter o cadena)**: devuelve la posición que ocupa la primera ocurrencia del carácter o cadena especificada como argumento. Si el carácter no se encuentra dentro de la cadena devuelve -1.
- Los métodos **replace(old, new)**, **replaceAll(old, new)** y **replaceFirst(old, new)**: estos métodos devuelven una nueva cadena resultado de reemplazar las ocurrencias de la subcadena *old* por la subcadena *new*.
- La función **split(delimitador)**: devuelve un *array* de objetos *String* que resulta de la división de la cadena del objeto al que se aplica en subcadenas divididas por el carácter especificado como argumento.
- El método **trim()**: devuelve una nueva cadena que resulta de recortar en la cadena original los espacios en blanco que haya al principio y al final.
- El método **substring(posición)**: extrae y devuelve una subcadena a partir de la posición especificada como argumento.
- El método **substring(inicio, fin)**: extrae y devuelve una subcadena desde la posición de *inicio* hasta la posición de *fin*.

# StringTokenizer

La clase *StringTokenizer* también representa a una cadena de caracteres como *String*, pero se utiliza en los casos en que necesitamos dividir la cadena en varias piezas o *tokens*.

Al construir un nuevo objeto de la clase *StringTokenizer* pasamos dos argumentos, el primero es la cadena de texto que queremos dividir y el segundo es el delimitador.

```
new StringTokenizer("Hola-Hello-Hallo-Ciao-Ni hao-salut-ahoj", "-");
```

Podemos recorrer los distintos *tokens* de la cadena con un algoritmo similar al de lectura secuencial de un fichero de texto, es decir, vamos leyendo *tokens* mientras existan más.

```
import java.util.StringTokenizer;

public class Principal {
    public static void main(String[] args) {
        StringTokenizer texto = new StringTokenizer("Hola-Hello-Hallo-Ciao-Ni hao-salut-ahoj", "-");
        System.out.println("Número de tokens: " + texto.countTokens());
        while (texto.hasMoreTokens()) {
            String subcadena = texto.nextToken();
            System.out.println(subcadena);
        }
    }
}
```

El programa anterior va leyendo secuencialmente *tokens* de la cadena con la siguiente sentencia:

`texto.nextToken()`

El proceso terminará cuando ya no existan más *tokens* que leer, en cuyo caso dejará de cumplirse la siguiente condición del *while*:

`texto.hasMoreTokens()`



El método `countTokens()` devuelve el número de *tokens* que tiene la cadena.

Observa que hemos tenido que añadir al principio del programa la siguiente línea:

```
import java.util.StringTokenizer;
```

Con esta línea hemos importado la clase *StringTokenizer* del paquete *java.util*, que es donde se encuentra la librería que la contiene. Más adelante comprenderás mejor el concepto de librería y paquete.

# StringBuilder

*StringBuilder*, igual que la clase *String*, permite la creación de objetos para manejar cadenas de caracteres, sin embargo, hay diferencias importantes que irás descubriendo con ejemplos.

## Diferencias entre *String* y *StringBuilder*:

- 1 Para construir un objeto *StringBuilder* tienes que usar el operador *new*, mientras que con *String* se puede omitir.
- 2 Los objetos de la clase *String* son inmutables, mientras que los objetos de la clase *StringBuilder* son mutables.
- 3 Los objetos de la clase *String*, al ser inmutables, no cambian su tamaño. Un objeto *StringBuilder* cambia su tamaño de almacenamiento en la memoria RAM para adaptarse a los cambios que va sufriendo.



### Pero, ¿qué es un objeto mutable?

Se dice que un objeto es mutable cuando al utilizar sus métodos cambiamos su estado, es decir, el valor de sus propiedades. En el caso de un objeto *StringBuilder* es mutable porque sus métodos cambian el texto original con que se construyó. Sin embargo, los métodos de un objeto *String* no modifican el texto que representa.

Hagamos una prueba para comprobar que los objetos de la clase *String* son inmutables.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto1 = "El perro de Roque no tiene rabo ";  
  
        texto1.toUpperCase();  
        texto1.concat("porque Ramon Ramirez se lo ha robado");  
  
        System.out.println(texto1);  
    }  
}
```

Has declarado un objeto de la clase *String* llamado *texto1*, luego lo has convertido a mayúsculas y después le has concatenado otro texto. Sin embargo, cuando muestras en pantalla el texto, observas que ninguno de estos cambios se ha realizado, ya que el texto sigue siendo el mismo.

Los métodos de la clase *String* no modifican el texto, lo que hacen es devolver otro texto con el resultado de la modificación. Como en el ejemplo no estamos recogiendo el resultado devuelto en ninguna variable, pues en realidad no hemos hecho nada.

Si queremos modificar el objeto la única manera es mediante una asignación.

```
public class Principal {  
    public static void main(String[] args) {  
        String texto1 = "El perro de Roque no tiene rabo ";  
  
        texto1 = texto1.toUpperCase();  
        texto1 = texto1.concat("porque Ramon Ramirez se lo ha robado");  
  
        System.out.println(texto1);  
    }  
}
```

En este otro ejemplo comprobarás que sí se realiza el cambio. Pero no ha sido obra de los métodos utilizados, sino por las asignaciones que estamos realizando. Los métodos de la clase *String* devuelven un nuevo objeto, que en este caso estamos asignando a la misma variable, y el anterior objeto pasará a ser gestionado por el recolector de basura de Java para que termine siendo liberado de la memoria RAM (concepto que aprenderás más adelante).

Hagamos ahora una prueba que demuestre que *StringBuilder* "sí" es mutable.

```
public class Principal {  
    public static void main(String[] args) {  
        StringBuilder texto1 = new StringBuilder("El perro de Roque no tiene rabo ");  
  
        texto1.append("porque Ramon Ramirez se lo ha robado"); // Añadir al final.  
        System.out.println(texto1);  
  
        texto1.delete(45, 53); // Eliminamos la subcadena Ramirez.  
        System.out.println(texto1);  
  
        texto1.insert(45, "Pérez "); // Insertar en la posición indicada.  
        System.out.println(texto1);  
    }  
}
```

Hemos utilizado sobre el objeto *StringBuilder* los siguientes métodos:

#### **append()**

##### **append()**

Concatena el objeto al que se aplica con el valor del argumento, que podrá ser de tipo *String*, *int*, *float*, *double*, etc. El método *append()* se encargará de realizar la conversión a texto cuando sea necesario.

#### **insert()**

##### **insert()**

Inserta en el objeto al que se aplica un valor en la posición indicada en el primer argumento. El valor insertado será el especificado en el segundo argumento, que puede ser de tipo *String*, *int*, *float*, *double*, etc. El método *insert()* se encargará de realizar la conversión a texto si es necesario.

#### **delete()**

##### **delete()**

Recorta del objeto al que se aplica una subcadena, que viene determinada por los dos argumentos. El primer argumento es la posición de inicio y el segundo argumento es la posición final.

Es posible construir un objeto *StringBuilder* a partir de un objeto *String* pasándoselo como argumento al constructor de *StringBuilder*:

```
String texto1 = "El perro de Roque no tiene rabo";  
StringBuilder texto2 = new StringBuilder (texto1);
```

También se puede obtener un objeto *String* a partir de un objeto *StringBuilder* utilizando el método *toString()*.

```
StringBuilder texto1 = new StringBuilder ("El perro de Roque no tiene rabo");  
String texto2 = texto1.toString();
```

---

**Los objetos *String*, al ser inmutables o estáticos, consumen menos recursos de memoria y son más óptimos siempre que su uso sea apropiado. Sin embargo, para cadenas muy dinámicas que deben cambiar mucho de tamaño es más adecuado utilizar *StringBuilder*.**

# StringBuffer

La clase *StringBuffer* funciona exactamente igual que la clase *StringBuilder*. La diferencia es que sus objetos son sincronizados, lo que significa que son más seguros para trabajar en aplicaciones multitarea.

Para que puedas comprobar que *StringBuffer* funciona igual que *StringBuilder* puedes probar el mismo ejemplo anterior pero cambiando el nombre de la clase:

```
public class Principal {  
    public static void main(String[] args) {  
        StringBuffer texto1 = new StringBuffer("El perro de Roque no tiene rabo ");  
  
        texto1.append("porque Ramon Ramirez se lo ha robado"); // Añadir al final.  
        System.out.println(texto1);  
  
        texto1.delete(45, 53); // Eliminamos la subcadena Ramirez.  
        System.out.println(texto1);  
  
        texto1.insert(45, "Pérez "); // Insertar en la posición indicada.  
        System.out.println(texto1);  
    }  
}
```

Todo funciona igual, ¿verdad? Sin embargo, si no vamos a trabajar con hilos y concurrencia, deberías seguir utilizando *StringBuilder*.



La concurrencia es una propiedad de algunos programas que tienen la capacidad de permitir la ejecución simultánea de varios procesos. Incluso es posible que los procesos que se ejecutan simultáneamente puedan comunicarse entre sí.

# Resumen

---

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Java no tiene un tipo de dato elemental para representar cadenas de caracteres. Para almacenar un texto debemos crear un objeto de una de las clases disponibles para el manejo de datos alfanúmericos o cadenas.
- La clase *String* representa una cadena de texto y sus objetos disponen de muchísimos métodos para manipularla. Algunos ejemplos son: *length()*, *substring()*, *split()*, *toUpperCase()*, etc.
- *StringTokenizer* también es una clase que sirve para crear cadenas de texto, pero solo se utiliza en los casos en que necesitamos dividir la cadena en varias piezas o *tokens*.
- La clase *StringBuilder* es muy similar a la clase *String*, pero es mutable, mientras que *String* es inmutable.
- Por último, la clase *StringBuffer* también es mutable y muy parecida a *StringBuilder*, pero es sincronizada, de manera que resulta más segura en aplicaciones multitarea.



**PROEDUCA**