3.1 Cadenas

1. Introducción

El texto es uno de los tipos de datos más importantes y por ello Java lo trata de manera especial. Para Java las cadenas de texto son objetos especiales porque en determinados aspectos funcionan como los primitivos y en otros aspectos funcionan como los objetos.

Las cadenas deben manejarse creando objetos de tipo **String**. Se pueden inicializar de dos maneras:

- Usando el operador asignación como los primitivos: `String s="hola";`
- Usando el constructor como los objetos: `String s=new String("hola");`

Los **literales** cadena se escriben entre comillas dobles: `"Esto es un literal cadena"`.

En java existe también la **cadena vacía ("")**, es decir, una cadena sin ningún carácter.

Ejemplo: `String s="";` A la variable s se le está asignando la cadena vacía.

Algunas veces, para comparar que una variable de tipo `String` es equivalente a una determinada constante de cadena se usa la construcción `"Hello".equals(message)`, ya que dicho construcción no puede lanzar `NullPointerException` si *message* es *null*, sino que tan sólo retornará *false*, mientras que `message.equals("Hello")` lanzaría `NullPointerException` en ese caso.

Los *Text Blocks* (Bloques de Texto) permiten definir cadenas de texto multilínea de manera más sencilla y legible. Los bloques de texto comienzan y terminan con tres comillas dobles `"""`. En el ejemplo se pueden observar dos maneras de definir la misma cadena donde la segunda (s2) está utilizando *Text Blocks*:

```
String s1 = "Línea 1\nLínea 2\nLínea 3";
String s2 = """
            Línea 1
            Línea 2
            Línea 3
            """;
```

Veamos otro ejemplo:

```
String s1 = """
            Línea 1
            Línea 2
            Línea 3
            """;
String s2 = """
                Línea 1
                Línea 2
                Línea 3
            """;
System.out.println(s1);
System.out.println(s2);
```

Salida por consola:

```
Línea 1
Línea 2
Línea 3

    Línea 1
    Línea 2
    Línea 3
```

El compilador escapará e interpretará los caracteres especiales de modo que no tenemos que usar la barra invertida en los *Text Blocks* para escapar.

2. Inmutabilidad

Uno de los conceptos que suele venir asociado a la programación funcional es el de la inmutabilidad. Si bien es cierto que la inmutabilidad es una idea que no es exclusiva de la programación funcional, sí que cobra una importancia vital en este tipo de lenguajes. La programación funcional se asienta sobre muchos conceptos matemáticos que requieren de la inmutabilidad para seguir siendo válidos. Aún así, es un concepto que es interesante conocer independientemente del tipo de paradigma de programación que se utilice.

¿Qué es la inmutabilidad?

La idea es muy sencilla de entender: algo es inmutable cuando no se puede modificar. En el contexto de la programación, una variable es inmutable cuando su valor no se puede modificar. Y un objeto lo es cuando su estado no puede ser actualizado tras la creación del objeto. Es por tanto una forma de asegurar que los objetos no se modifican en lugares inesperados afectando con ello la ejecución de nuestro programa.

La inmutabilidad genera muchas ventajas en las aplicaciones multihilo, donde la inmutabilidad simplifica mucho el tratamiento de la concurrencia. Si algo no se puede modificar, da igual que se acceda a ello desde distintos hilos a la vez, así como el orden en que se haga.

La inmutabilidad hace que el código sea mucho más predecible y más fácil de testear, porque se acota mucho más los lugares donde se producen modificaciones de estado.

La inmutabilidad tiene el sobrecoste de la generación de objetos nuevos cada vez que cambia el estado, así que esto puede penalizar bastante en el rendimiento. Así como guía muy general, es recomendable usar objetos mutables en cualquier situación que requiera de un estado que se modifica a menudo y/o la duplicación de ese estado sea costosa.

La inmutabilidad, por el contrario, presenta ventajas en las siguientes situaciones:

- En objetos que no vayan a requerir modificaciones de estado.
- En objetos que sean simples de duplicar.
- En situaciones de concurrencia.

Así que hay que hacer inmutable todo el código que sea posible ya que es un concepto muy potente que nos puede ayudar a simplificar la complejidad de comprensión de nuestro código y a disminuir las probabilidades de que se produzcan errores inesperados.

En Java, los objetos String son inmutables.

3. Formatos de codificación de caracteres

En el Unicode, el Plano Multilingüe Básico (Basic Multilingual Plane, ***BMP***) incluye los 65.536 caracteres cuyos códigos van desde U+0000 a U+FFFF, que son la mayoría de los caracteres utilizados más frecuentemente.

El número 65.536 es 2 elevado a 16, es decir, la cantidad máxima de combinaciones de bits que se pueden obtener en dos bytes.

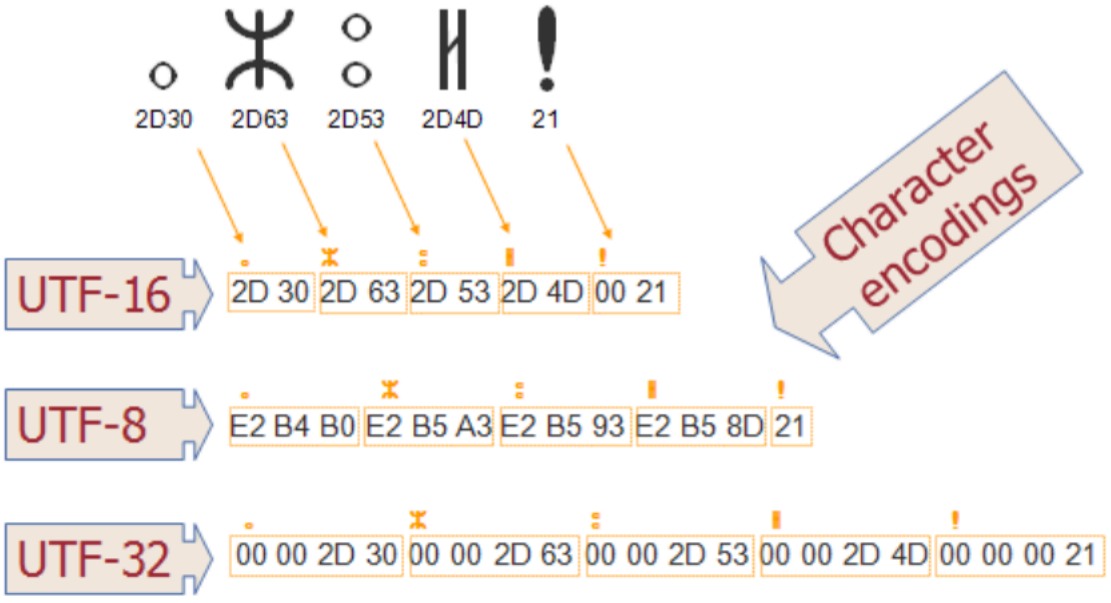
El resto de caracteres se denominan caracteres complementarios.

Los formatos de codificación que se pueden usar con Unicode se denominan UTF-8, UTF-16 y UTF-32:

UTF-8 utiliza 1 byte para representar caracteres ASCII, dos bytes para caracteres en otros bloques alfabéticos y tres bytes para el resto del BMP. Para los caracteres complementarios se utilizan 4 bytes.

UTF-16 utiliza 2 bytes para cualquier carácter en el BMP y 4 bytes para los caracteres complementarios.

UTF-32 emplea 4 bytes para todos los caracteres.



En Java, las cadenas utilizan el formato de codificación UTF-16.

4. Formateo de cadenas

Hay un método estático en la clase String que sirve para construir una cadena con cierto formato. Funciona de la misma manera que el método `printf` que vimos en el *Tema 1.4 Escritura en pantalla*.

```
package tema3_1_Cadenas;

public class StringClass1 {

    public void show() {

        String formattedString;
        formattedString = String.format("Nombre: %s Edad: %d Sueldo: %.2f",
"Juan", 20, 1896.23);
        System.out.println(formattedString);

    }

    public static void main(String[] args) {

        new StringClass1().show();

    }

}
```

5. Métodos de la clase String

Si observamos en la API la clase String, incluye métodos para examinar los caracteres individuales de una cadena, para comparar cadenas, para buscar cadenas, para extraer subcadenas, para convertir cadenas a mayúsculas o minúsculas, etc.

Como vimos en el *Tema 1.1 Introducción al lenguaje Java*, el operador concatenación `+` es un operador binario que devuelve una cadena resultado de concatenar las dos cadenas que actúan como operandos. Si sólo uno de los operandos es de tipo cadena, el otro operando se convierte implícitamente en tipo cadena.

Obsérvese en la API el método **valueOf**: es estático y está sobrecargado. Sirve para obtener la representación String de un valor u objeto.

```
package tema3_1_Cadenas;

public class StringClass2 {

    public void show() {

        int i = 100;
        String string1, string2, string3, string4;

        string1 = "Esto es un literal cadena"; //Se le da un valor inicial a la
cadena con el operador de asignación =
        System.out.println(string1);
```

```
        System.out.println(string1 + " al cual le hemos concatenado este literal
cadena"); //Se concatena otra cadena con el operador +

        string2 = "hola";
        string3 = " que tal";
        string4 = string2 + string3;
        System.out.println(string4);

        System.out.println(i + 100); //Suma de enteros
        System.out.println(String.valueOf(i) + 100); //Concatenación de cadenas

    }

    public static void main(String[] args) {

        new StringClass2().show();

    }

}
```

Otros métodos de las cadenas muy útiles son:

- **charAt**: devuelve el carácter de la cadena del especificado índice. Dicho índice empieza en cero, es decir, con el cero se obtiene el primer carácter de la cadena.
- **length**: devuelve la longitud de la cadena.
- **equals**: compara si dos cadenas son iguales. Las cadenas se comparan con equals como los objetos y los primitivos se comparan con ==.
- **equalsIgnoreCase:** hace lo mismo que el anterior pero no tiene en cuenta las mayúsculas y minúsculas.

```
package tema3_1_Cadenas;

public class StringClass3 {

    public void show() {

        String string = "hola";
        System.out.println(string.charAt(0));//h
        System.out.println(string.charAt(1));//o
        System.out.println(string.charAt(2));//l
        System.out.println(string.charAt(3));//a
        System.out.println(string.length());//4
        System.out.println(string.equals("hola"));//true
        System.out.println(string.equals("Hola"));//false
        System.out.println(string.equals("adiós"));//false
        System.out.println(string.equalsIgnoreCase("Hola"));//true
        System.out.println(string.equalsIgnoreCase("HOLA"));//true

        //También se le pueden aplicar métodos a un literal cadena:
        System.out.println("hola".equals("hola")); //true
        System.out.println("adios".equals("hola"));//false

    }
```

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

        new StringClass3().show();

    }

}
```

Más métodos de cadenas interesantes:

- **compareTo**: compara dos cadenas lexicográficamente. La comparación se basa en el valor Unicode de cada carácter de las cadenas.

  `s1.compareTo(s2)`: devuelve un número negativo si s1 es menor, un número positivo si s1 es mayor o cero si son iguales.
- **compareToIgnoreCase**: hace lo mismo que el anterior pero no tiene en cuenta las mayúsculas y minúsculas**.**
- **concat:** se utiliza para concatenar cadenas, como el operador +.
- **endsWith:** devuelve true si la cadena termina con un determinado texto.
- **indent**: ajusta la sangría de una cadena de texto, es decir, permite añadir o quitar espacios en blanco al principio de cada línea de una cadena.
- **startsWith**: devuelve true si la cadena empieza con un determinado texto.
- **indexOf:** devuelve la primera posición en la que aparece un determinado texto en la cadena. En el caso de que el texto buscado no se encuentre, devuelve -1. Este método está sobrecargado para que el texto a buscar pueda ser char o String.
- **lastIndexOf:** es como el anterior pero busca desde el final.
- **isEmpty:** devuelve true si la cadena está vacía, es decir, si su longitud es cero.
- **regiontMatches**: compara una parte específica (una región) de una cadena con una parte específica de otra cadena. No necesariamente compara las cadenas enteras, sino solo las subcadenas seleccionadas por los índices y la longitud proporcionados. Se puede elegir si la comparación es sensible o no a mayúsculas y minúsculas.
- **repeat:** devuelve una cadena cuyo valor es la concatenación de la cadena repetida varias veces.
- **replace**: reemplaza todas las apariciones de un texto por otro texto. Este método está sobrecargado para que el texto pueda ser char o String.
- **substring**: obtiene una subcadena.
- **toLowerCase**: devuelve la cadena en minúsculas.
- **toUpperCase**: devuelve la cadena en mayúsculas.
- **trim**: elimina los espacios en blanco del principio y del final de la cadena.

```
package tema3_1_Cadenas;

public class StringClass4 {

    public void show() {

        String string1 = "hola", string2 = "adios", string3 = "Hola";
```

```
        String string5 = "Estoy aprendiendo Programación";
        String string6 = "Hace tiempo estuve aprendiendo Junit";
        String string7 = """
                         Línea 1
                         Línea 2
                         Línea 3
                         """;
        String string8 = "    Línea 1\n    Línea 2\n    Línea 3";
        String string4,string9,string10;

        //hola es mayor que adios
        System.out.println(string1.compareTo(string2) > 0 ? String.format("%s es
mayor que %s", string1, string2)
                : String.format("%s es menor que %s", string1, string2));
        //adios es menor que hola
        System.out.println(string2.compareTo(string1) > 0 ? String.format("%s es
mayor que %s", string2, string1)
                : String.format("%s es menor que %s", string2, string1));
        //En el Unicode, las mayúsculas están antes: Hola es menor que adios
        System.out.println(string3.compareTo(string2) > 0 ? String.format("%s es
mayor que %s", string3, string2)
                : String.format("%s es menor que %s", string3, string2));
        //Hola es mayor que adios si no tenemos en cuenta las mayúsculas y
minúsculas
        System.out.println(string3.compareToIgnoreCase(string2) > 0
                ? String.format("%s es mayor que %s si no tenemos en cuenta las
mayúsculas y minúsculas", string3,
                        string2)
                : String.format("%s es menor que %s si no tenemos en cuenta las
mayúsculas y minúsculas", string3,
                        string2));

        System.out.println(string4 = string1.concat(" que tal"));//string4="hola
que tal"
        System.out.println(string4.endsWith("tal"));//true
        System.out.println(string4.endsWith("hola"));//false
        System.out.println(string4.startsWith("hola"));//true
        System.out.println(string4.startsWith("tal"));//false

        System.out.println(string4.indexOf("hola"));//0
        System.out.println(string4.indexOf("tal"));//9
        System.out.println(string4.indexOf("que"));//5
        System.out.println(string4.indexOf(string2));//-1
        System.out.println(string4.indexOf(string3));//-1
        System.out.println(string4.indexOf('a'));//3
        System.out.println("hola que tal hola que tal".lastIndexOf("tal"));//22

        System.out.println(string4.isEmpty());//false
        System.out.println("".isEmpty());//true

        System.out.println("ole ".repeat(6));//ole ole ole ole ole ole

        System.out.println(string4.replace('a', '*'));//hol* que t*l
        System.out.println("hola que tal hola que tal".replace("hola",
"buenas"));//buenas que tal buenas que tal
```

```
        //Para hacer desaparecer partes de una cadena, se reemplazan por cadena
vacía
        System.out.println("hola que tal hola que tal".replace(" ",
""));//holaquetalholaquetal

        System.out.println(string4.substring(9));//tal
        System.out.println(string4.substring(5, 8));//que

        System.out.println("HOLA QUE TAL".toLowerCase());//hola que tal
        System.out.println("hola que tal".toUpperCase());//HOLA QUE TAL
        System.out.println("       Hola que tal      ".trim());//Hola que tal
        System.out.printf("Las regiones %s son iguales\n",
string5.regionMatches(true, 6, string6, 19, 11)?"sí":"no");

        //Se añaden 4 espacios al principio de cada línea
        string9 = string7.indent(4);
        System.out.printf("Con sangría añadida:\n%s",string9);

        //Se quitan 2 espacios al principio de cada línea
        string10 = string8.indent(-2);
        System.out.printf("Con sangría eliminada:\n%s",string10);

    }

    public static void main(String[] args) {

        new StringClass4().show();

    }

}
```

6. La interfaz CharSequence

Tal y como se explica en el Tema 7. Herencia 12. Interfaces, una interfaz sirve para especificar un comportamiento que ciertas clases pueden seguir sin dictar cómo se debe implementar ese comportamiento. Pues la interfaz `CharSequence`, es una representación legible de una secuencia de caracteres. Está presente en el paquete `java.lang` y define un conjunto de métodos para trabajar con secuencias de caracteres de una manera uniforme. Sus métodos principales son `length`, `charAt` y `subSequence`. Las clases más comunes que implementan esta interfaz son `String`, `StringBuilder` y `StringBuffer`.

Ejemplo:

```
package tema3_1_Cadenas;

public class InterfaceCharSequence {

    public void show() {

        CharSequence cs = "Estoy aprendiendo a programar en Java";
        System.out.printf("Length: %d\n", cs.length());
        System.out.printf("Char at 1: %c\n", cs.charAt(1));
        System.out.printf("Subsequence (0, 5): %s",cs.subSequence(0, 5));
```

```
    }

    public static void main(String[] args) {

        new InterfaceCharSequence().show();

    }

}
```

Salida por consola:

```
Length: 37
Char at 1: s
Subsequence (0, 5): Estoy
```

7. La clase StringBuilder

A diferencia de `String` que es inmutable (no puede cambiar una vez creada), `StringBuilder` permite modificar el contenido de la cadena sin crear nuevos objetos, lo que lo hace más eficiente en términos de memoria y rendimiento cuando se realizan múltiples modificaciones.

Características principales:

- **Mutable**: puede modificar el contenido sin crear un nuevo objeto.
- **Eficiente**: mejor rendimiento en operaciones como concatenación, inserción o borrado de caracteres.
- **Sincronización**: no es thread-safe, es decir, no es segura para su uso en entornos con múltiples hilos. Java dispone de la clase `StringBuffer` que sí es thread-safe, aunque a costa de un rendimiento ligeramente inferior en comparación con `StringBuilder`.

Métodos comunes:

- **append**: añade texto al final.
- **insert**: inserta texto en una posición específica.
- **delete**: elimina caracteres.
- **reverse**: invierte la secuencia de caracteres.

Ejemplo:

```
package tema3_1_Cadenas;

public class StringBuilderClass {

    public void show() {

        StringBuilder sb = new StringBuilder("Estoy aprendiendo");
        System.out.println(sb);
        sb.append(" Java");
        System.out.printf("Después de append: %s%n", sb);
        sb.insert(17, " a programar en");
        System.out.printf("Después de insert: %s%n", sb);
```

```
        sb.delete(17, 32);
        System.out.printf("Después de delete: %s%n", sb);
        sb.reverse();
        System.out.printf("Después de reverse: %s%n", sb);

    }

    public static void main(String[] args) {

        new StringBuilderClass().show();

    }

}
```