Unidad 5 Clases Especiales en Java

Índice

1 Actividad Inicial. Explicación de la API java.

1. Se le enseñará un ejemplo a los alumnos de una aplicación que hace uso de muchas librerías.
2. Es una aplicación springboot con acceso a datos JPA. A continuación se introducirá la API java.

2 Conceptos de API java, JRE y java.base.

2.1 Las API´s de java.

La **API Java es una interfaz de programación de aplicaciones** (API, por sus siglas del inglés: **Application Programming Interface**) provista por los **creadores del lenguaje de programación Java**, que **proporciona a los programadores los medios para desarrollar aplicaciones** Java.
Como **el lenguaje Java es un lenguaje orientado a objetos**, la **API de Java provee de un conjunto de clases** utilitarias para efectuar toda clase de tareas necesarias dentro de un programa.
La **API Java está organizada en paquetes lógicos**, donde **cada paquete contiene un conjunto de clases relacionadas** semánticamente.
Existen numerosas **API de Java para realizar todo tipo de operaciones**,1 algunas de las más conocidas son:

1. **JAXP**: para procesar **XML**.
2. **Servlets**: para facilitar la implementación de **soluciones web**.
3. **Hibernate**: para facilitar la **implementación de persistencia**.

La **función de las API's de java** es ofrecernos **herramientas a los desarrolladores** para crear aplicaciones. Nos hacen la vida más fácil, **permitiéndonos reutilizar código**, en lugar de tener que programarlo nosotros. Utilidades como ordenar, tener clasificado, hacer **operaciones matemáticas**, hacer **búsquedas de texto**, pedir **datos al usuario** y **muchos más procesos** son cuestiones **que se** repiten con frecuencia en programación, y por tanto se encontrarán resueltas en el API de Java. Por supuesto que **podemos crear algoritmos propios para ordenar listas**, pero lo **más rápido y eficiente en general será usar las herramientas del API** disponibles porque están desarrolladas por profesionales y han sido depuradas y optimizadas a lo largo de los años y versiones del lenguaje.

2.2 Java.base, java.lang y javax

Java.base **define las API fundamentales de java SE Platform**, y por tanto la base para **desarrollar aplicaciones que corran sobre la JRE**, que contiene la JVM. Proveedores: **la JDK** es un módulo que **proporciona una implementación**

del proveedor del sistema de **archivos jrt para enumerar y leer los archivos de clase y recursos** en una imagen en **tiempo de ejecución**. Es la que nos **permite crear y gestionar ficheros de clases y recursos, compilar, ejecutar** nuestros programas. Es la base sobre la que se desarrollan más API's y aplicaciones.

java.lang. Proporciona **clases que son fundamentales para el diseño del lenguaje de programación Java**. Las clases más importantes son **Object** , que es la raíz de la jerarquía de **clases y Class** , instancias de las cuales **representan clases en tiempo de ejecución.** Nos permite **crear nuevas clases y objetos, herencia y polimorfismo**.

Estas **dos librerias no hace falta importarlas**, ni requerirlas en modulos, son **importadas de manera implicita**. En cualquier caso, **todas las librerias java van organizadas en paquetes** al igual que **debemos organizar nuestros componentes de nuestras aplicaciones en paquetes**. Por ejemplo, para la librería de ficheros, en la **documentación se referenciarán** como **API java.io** o **package java.io.**

El lenguaje de programación Java utiliza el **prefijo javax** para un **paquete de extensiones Java estándar**. Estos incluyen **extensiones como javax.servlet**, que se ocupa de la **ejecución de servlets. La API jcr**, que se ocupa de la **biblioteca de contenido Java** y acceso a **repositorios como CMS**. La **API swing** que se ocupan de **los interfaces de escritori**o, etc. En resumen, **se usa para extensiones** como Java Enterprise, desarrollo para aplicaciones empresariales.

2.3 Librerías básicas java, ampliando conocimientos

Cuando tengamos **experiencia como programadores Java**, posiblemente **dispongamos de clases desarrolladas por nosotros mismos** que utilicemos en distintos proyectos. En **empresas grandes**, es **frecuente disponer de clases desarrolladas por compañeros** de la empresa que **usaremos de forma parecida a como se usa el API de Java**: **conociendo su interfaz** pero no su implementación. Trabajar **con una clase sin ver su código fuente requiere que exista una buena documen**tación que nos sirva de guía. Hablaremos de la **documentación de las clases y proyectos en Java** un poco más adelante. De momento, vamos a **aprender a usar la documentación del API de Java**.

En primer lugar, debemos **tener una idea de cómo se organizan las clases del API.** Esta **organización es en forma de árbol jerárquico**, como se ve en la figura "Esquema orientativo de la organización de librerías en el API de Java". Esta **figura trata de mostrar la organización del API de Java**, pero **no recoge todos los paquetes ni clases existentes** que son muchos más y no cabrían ni en una ni en varias hojas.

Los **nombres de las librerías** responden a este esquema jerárquico y **se basan**

en la notación de punto. Por ejemplo, el **nombre completo** para la **clase ArrayList** sería **java.util.ArrayList**. Se permite el **uso de * para nombrar a un conjunto de clases**. Por ejemplo **java.util.*** hace referencia **al conjunto de clases** dentro del **paquete java.util**, donde tenemos ArrayList, LinkedList y otras clases.

Para **utilizar las librerías del API**, existen **dos situaciones**:

a) Hay **librerías o clases que se usan siempre** pues constituyen **elementos fundamentales del lenguaje Java como** la clase **String**. Esta clase, perteneciente al **paquete java.lang**, se puede utilizar directamente en cualquier programa Java ya que **se carga automáticamente.**

b) Hay **librerías o clases que no siempre se usan**. Para **usarlas dentro de nuestro código hemos de indicar** que **requerimos su carga mediante** una sentencia **import incluida en cabecera de clase**. Por ejemplo import java.util.ArrayList; es una sentencia que incluida en cabecera de una clase nos permite usar la clase ArrayList del API de Java. Escribir **import java.util.*; nos permitiría cargar todas las clases del paquete java.util.** Algunos paquetes **tienen decenas o cientos de clases**. Por ello nosotros **preferiremos en general especificar las clases** antes que usar asteriscos ya que evita la carga en memoria de clases que no vamos a usar. Una **clase importada se puede usar de la misma manera que si fuera una clase generada por nosotros**: **podemos crear objetos de esa clase y llamar a métodos** para operar sobre esos objetos. Además, cada **clase tendrá uno o varios constructores**.

Encontrar **un listado de librerías o clases más usadas es una tarea casi imposible**. Cada programador, dependiendo de su actividad, utiliza ciertas librerías que posiblemente no usen otros programadores. Los **programadores centrados en programación de escritorio usarán clases diferentes** a las que usan **programadores web** o de gestión de bases de datos. Las clases y las librerías básicas deberás ir conociéndolas mediante **cursos o lecciones de formación** en **Java**. Las **clases y librerías más avanzadas deberás utilizarlas y estudiarlas a medida** que te vayan **siendo necesarias para el desarrollo de aplicaciones**, ya que su estudio completo es prácticamente imposible. Podemos citar clases de uso amplio.

En el **paquete java.io: clases File, Filewriter, Filereader**, etc. En el paquete **java.lang:** clases **System, String, Thread**, etc. En el paquete **java.security**: clases que permiten implementar **encriptación y seguridad**. En el **paquete paquete java.util: clases ArrayList, LinkedList, HashMap, HashSet, TreeSet, Date, Calendar, StringTokenizer, Random**, etc. En los paquetes **java.awt y javax.swing una biblioteca gráfica**: desarrollo de **interfaces gráficas** de usuario con ventanas, botones, etc.
Insistimos en una idea: **no trateis de memorizar la organización detallada del API de Java** ni un **listado de clases más usadas** porque esto **tiene poco sentido**. Lo importante es **que conozcáis la forma de organización, cómo se** estructuran y utilizan las clases y que aprendas a buscar información para encontrarla rápidamente cuando te sea necesaria.

Para programar en Java tendremos que recurrir continuamente a consultar la documentación del API de Java. Esta **documentación está disponible en cds de libros y revistas especializadas o en internet** tecleando en un buscador como yahoo, google o bing el texto "api java 8" (o "api java 6", "api java 10", etc.) según la versión que estés utilizando. La **documentación del API de Java en general es correcta y completa**. Sin embargo, en casos excepcionales puede estar incompleta o contener erratas.

Mucha de la **documentación la encontrareis en la página oficial de Oracle**. Para documentar **las API's se usa habitualmente la aplicación Javadoc de la JDK que nos permite generar documentación a** partir de **etiquetas, comentarios y ficheros de texto**, genera de manera automática la documentación.

El sitio oficial de Oracle para java es:
https://www.oracle.com/java/

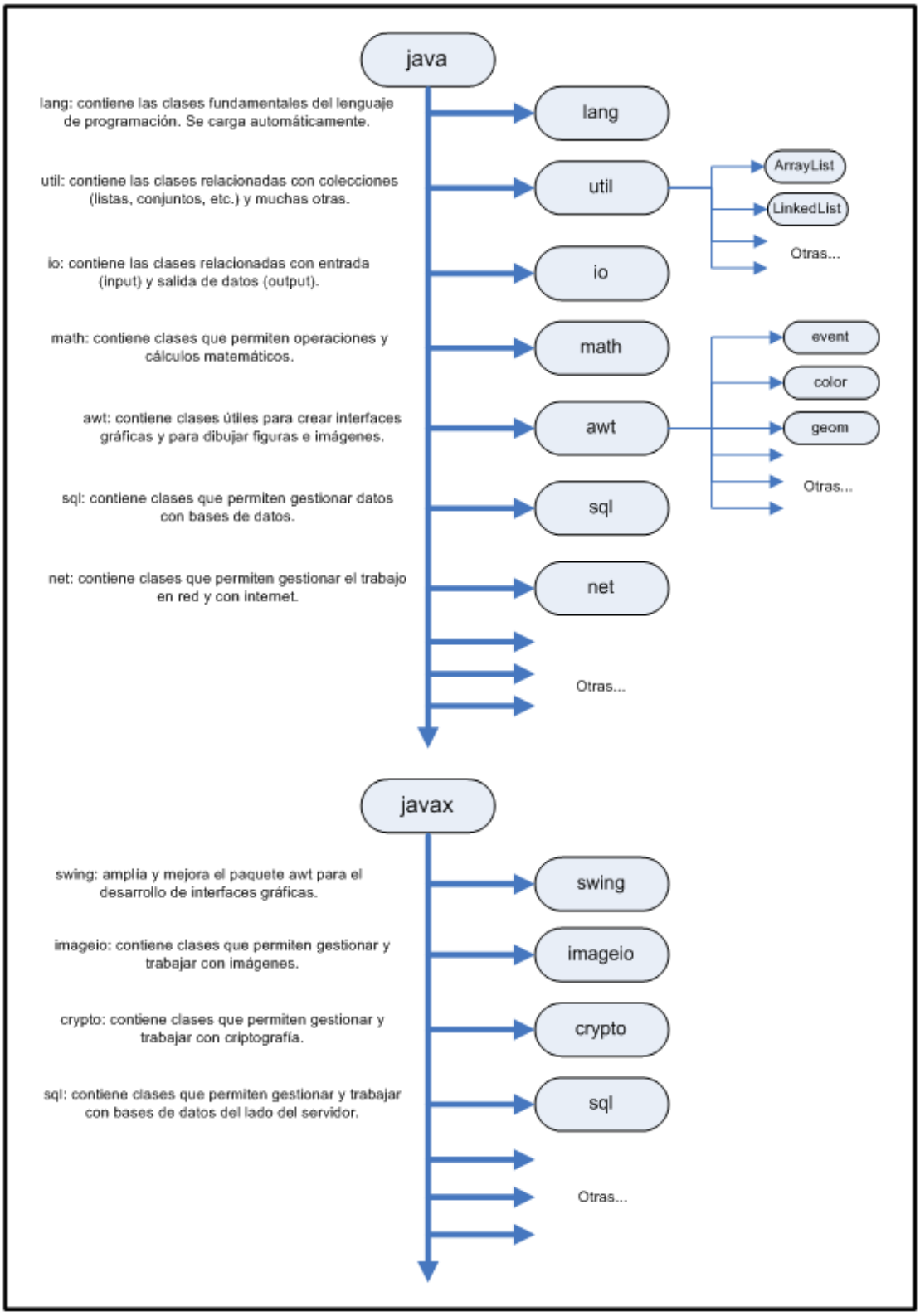
Podéis **encontrar información sobre descargas, API`s y más**. Es **nuestra primera fuente de documentación como programadores**, y es de obligada consulta, pues es la documentación oficial. **Java JSE es la versión oficial de Java desarrollada por Oracle**, que nos ofrece entre **otras cosas la JRE y la JDK,** kit de desarrollo para Java de Oracle. La documentación esta generada con javadoc.

https://docs.oracle.com/en/java/javase/14/docs/api/index.html
https://www.oracle.com/java/technologies/javase-downloads.html

Encontrareis **información extra en openJDK**. Es una **plataforma abierta para desarrolladores** de java. Ofrece una **implementación alternativa a la versión oficial** de java que ofrece en la JSE

https://openjdk.java.net/

En la siguiente imagen **teneis un gráfico resumén de las librerias** más utilizadas.


java
lang: contiene las clases fundamentales del lenguaje de programación. Se carga automáticamente.
lang
util: contiene las clases relacionadas con colecciones (listas, conjuntos, etc.) y muchas otras.
util
ArrayList
LinkedList
Otras...
io: contiene las clases relacionadas con entrada (input) y salida de datos (output).
io
math: contiene clases que permiten operaciones y cálculos matemáticos.
math
awt: contiene clases útiles para crear interfaces gráficas y para dibujar figuras e imágenes.
awt
event
color
geom
Otras...
sql: contiene clases que permiten gestionar datos con bases de datos.
sql
net: contiene clases que permiten gestionar el trabajo en red y con internet.
net
Otras...
javax
swing: amplía y mejora el paquete awt para el desarrollo de interfaces gráficas.
swing
imageio: contiene clases que permiten gestionar y trabajar con imágenes.
imageio
crypto: contiene clases que permiten gestionar y trabajar con criptografía.
crypto
sql: contiene clases que permiten gestionar y trabajar con bases de datos del lado del servidor.
sql
Otras...

3 Manejo de fechas en Java

LA clase Date en java dentro del paquete java.util nos permitía el manejo de tipo fecha en Java. Esta en desuso y a partir de Java 8 usaremos estas tres clases para manejar fechas y horas: **LocalDate, LocalTime y LocalDateTime.**

3.1 La clase LocalDate

Vamos a empezar con la clase LocalDate, donde podemos trabajar con nuestra hora local. Representa una fecha sin una zona horaria en el sistema de calendario ISO-8601, como 2007-12-03.

LocalDate es un objeto de fecha y hora inmutable que representa una fecha, a menudo vista como año-mes-día. También se puede acceder a otros campos de fecha, como día del año, día de la semana y semana del año. Por ejemplo, el valor "2nd October 2007" se puede almacenar en un LocalDate.

Esta clase no almacena ni representa una hora o zona horaria. En cambio, es una descripción de la fecha, como se usa para los cumpleaños. No puede representar un instante en la línea de tiempo sin información adicional, como un desplazamiento o una zona horaria, no podemos convertirlo a milisegundos.

Vamos algunos de sus métodos con ejemplos.

Creamos una fecha con la fecha actual con el método estático now.

```
LocalDate date= LocalDate.now();


System.out.println(date);
```

Como resultado obtendremos la siguiente salida:

```
2021-09-06
```

La creamos usando el método of con parámetros mes, año, día. Además, usamos tres métodos de LocalDate

```
LocalDate date2 = LocalDate.of(2021, 5, 9);

System.out.println("dia " + date2.getDayOfMonth() + " mes " +
date2.getMonthValue() + " año " + date2.getYear());
```

Parseando en formato ISO. Como veis escribimos la fecha en formato yyyy-MM-dd, año mes y día separado por guiones.

```
LocalDate date3 = LocalDate.parse("2021-08-14");
```

Opciones para los patrones

G	era	AD; Anno Domini; A
u	year	2004; 04
y	year-of-era	2004; 04
D	day-of-year	189
M/L	month-of-year	7; 07; Jul; July; J
d	day-of-month	10
Q/q	quarter-of-year	3; 03; Q3; 3rd quarter
Y	week-based-year	1996; 96
w	week-of-week-based-year	27
W	week-of-month	4
E	day-of-week	Tue; Tuesday; T
e/c	day-of-week	02,03
F	week-of-month	3
a	am-pm-of-day	PM o AM
h	clock-hour-of-am-pm	12
K	hour-of-am-pm	0-12
k	clock-hour-of-am-pm	0
H	hour-of-day	0-24
m	minute-of-hour	30
s	second-of-minute	55
S	fraction-of-second	976, 340
A	milli-of-day	1234
n	nano-of-second	987654321
N	nano-of-day	1234000000
V	time-zone	America/Los_Angeles; Z; -08:30
z	time-zone	Pacific Standard Time; PST
O	localized	GMT+8; GMT+08:00; UTC-08:00;
X	zone-offset	Z; -08; -0830; -08:30; -083015; -08:30:15;

Ejemplos:

Dia mes y año.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");

                // create a LocalDate object and
                LocalDate date4
                    = LocalDate.parse("06-09-2021", formatter);

            System.out.println(date4);
```

Dia de la semana, semana del mes, mes y año.

```
                 formatter = DateTimeFormatter.ofPattern("ee-W-MM-yyyy");

                // create a LocalDate object and
                LocalDate dateMes
                    = LocalDate.parse("07-2-09-2021", formatter);

            System.out.println(dateMes);
```

Dia del año y año.

```
                formatter = DateTimeFormatter.ofPattern("DD-yyyy");

        // create a LocalDate object and
        LocalDate date5
            = LocalDate.parse("123-2021", formatter);
```

Operando con fechas

Comparando con compareTo

```
LocalDate date= LocalDate.now();


            System.out.println(date);

            LocalDate date2 = LocalDate.of(2021, 5, 9);
```

```
            System.out.println("Comparando date y date2:" +
date.compareTo(date2));
```

```
Comparando date y date2:4
```

Viendo los dias de diferencias entre dos fechas. Usamos las clase Days con sus métodos estáticos. Realizamos el import y obtenemos la distancia entre las dos fechas con el método between.

```
import static java.time.temporal.ChronoUnit.DAYS;
```

```
long dias = DAYS.between(date3, date4);
```

```
        System.out.println("El Numero de dias entre: " + date3+ " y " + date4
+ " es: "+ dias); // 365 dias
```

```
El Numero de dias entre: 2021-08-14 y 2021-09-06 es: 23
```

Restamos dias a la fecha date3 con el método minusDays

```
      LocalDate date3 = LocalDate.parse("2021-08-14");
```

```
            System.out.println("fecha 3" + date3 + " fecha 3 menos 200
dias:" + date3.minusDays(200));
```

```
fecha 32021-08-14 fecha 3 menos 200 dias:2021-01-26
```

Sumamos 8 meses a date5, obteniendo la nueva fecha date6, con el método plusMonths.

```
        // create a LocalDate object and
        LocalDate date5
            = LocalDate.parse("123-2021", formatter);

        LocalDate date6= date5.plusMonths(8);
```

```
        System.out.println("Fecha con dia en el año:" + date5 + " le sumamos
8 meses " + date6);
```

```
Fecha con dia en el año:2021-05-03 le sumamos 8 meses 2022-01-03
```

Ejemplo operarConFechas

```
package fechas;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import static java.time.temporal.ChronoUnit.DAYS;

public class OperarConFechas {


public static void main(String[] args) {

            LocalDate date= LocalDate.now();


            LocalDate date2 = LocalDate.of(2021, 5, 9);

            System.out.println("Comparando date y date2:" +
date.compareTo(date2));

            LocalDate date3 = LocalDate.parse("2021-08-14");


            System.out.println("fecha 3" + date3 + " fecha 3 menos 200
dias:" + date3.minusDays(200));

             DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-
MM-yyyy");

              // create a LocalDate object and
              LocalDate date4
                  = LocalDate.parse("06-09-2021", formatter);

              long dias = DAYS.between(date3, date4);

              System.out.println("El Numero de dias entre: " + date3+ " y "
+ date4 + " es: "+ dias); // 365 dias

               formatter = DateTimeFormatter.ofPattern("ee-W-MM-yyyy");

              // create a LocalDate object and
              LocalDate dateMes
                  = LocalDate.parse("07-2-09-2021", formatter);

            System.out.println(dateMes);
```

```
            formatter = DateTimeFormatter.ofPattern("DD-yyyy");

        // create a LocalDate object and
        LocalDate date5
            = LocalDate.parse("123-2021", formatter);

        LocalDate date6= date5.plusMonths(8);


        System.out.println("Fecha con dia en el año:" + date5 + " le sumamos
8 meses " + date6);



      }


}
```

3.1.1 La case LocalTime

A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30.
LocalTime es un objeto de fecha y hora inmutable que representa una hora, a menudo vista como hora-minuto-segundo. El tiempo se representa con una precisión de nanosegundos. Por ejemplo, el valor "13:45.30.123456789" se puede almacenar en un LocalTime.

Esta clase no almacena ni representa una fecha o zona horaria. En cambio, es una descripción de la hora local como se ve en un reloj de pared. No puede representar un instante en la línea de tiempo sin información adicional, como un desplazamiento o una zona horaria.

El sistema de calendario ISO-8601 es el sistema de calendario civil moderno utilizado hoy en día en la mayor parte del mundo. Esta API asume que todos los sistemas de calendario usan la misma representación, esta clase, para la hora del día.

Esta es una clase basada en valores; el uso de operaciones sensibles a la identidad (incluida la igualdad de referencia (==), el código hash de identidad o la sincronización) en instancias de LocalTime puede tener resultados impredecibles y debe evitarse. El método de iguales debe utilizarse para las comparaciones.

Vamos a ver unos cuantos ejemplos de uso de esta clase para manejar horas.

Obtenemos la hora actual con el método estático now que accede al reloj del sistema.

```
LocalTime time= LocalTime.now();

            System.out.println("Hora actual:" + time);
```

```
Hora actual:10:06:59.452846900
```

Introducimos una hora usando el método of con horas, minutos y segundos.

```
LocalTime time2 = LocalTime.of(9, 12, 3);

          System.out.println("Hora con horas minutos y segundos:" + time2);
```

```
Hora con horas minutos y segundos:09:12:03
```

Usamos los métodos get para obtener horas, minutos, segundos y nanosegundos.

```
          System.out.println("Hora con horas"
                  + time2.getHour() + " minutos " + time2.getMinute() + " y
segundos " + time2.getSecond() +
                  "y nanosegundos " + time2.getNano());
```

Usamos un DateTimeFormatter para crear la hora con patrones. En este caso H m s, son Horas de 0 a 24, minutos (0-59), segundos (0-59).

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("H m s");
                LocalTime time5 = LocalTime.parse("14 30 40", formatter);


            System.out.println("Local time parseado con formatter:"  +
time5);
```

ManejoLocalTime.java

```
package fechas;

import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class ManejoLocalTime {


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

```
            LocalTime time= LocalTime.now();

            System.out.println("Hora actual:" + time);

            LocalTime time2 = LocalTime.of(9, 12, 3);

          System.out.println("Hora con horas minutos y segundos:" + time);

          System.out.println("Hora con horas"
                  + time2.getHour() + " minutos " + time2.getMinute() + " y
segundos " + time2.getSecond() +
                  "y nanosegundos " + time2.getNano());

            LocalTime time3 = LocalTime.of(9, 12, 3,100000);

            System.out.println("Hora con horas minutos, segundos y
nanosegundos:" + time3);


            LocalTime time4 = LocalTime.parse("10:15:45");

            System.out.println("Local time parseado: " + time4);


             DateTimeFormatter formatter = DateTimeFormatter.ofPattern("H m
s");
                LocalTime time5 = LocalTime.parse("14 30 40", formatter);


            System.out.println("Local time parseado con formatter:"  +
time5);




      }

}
```

3.1.2 Operando con la clase LocalTime

Vamos a operar con horas usando algunos de los métodos de LocalTime en el siguiente ejemplo.

Primero comparamos time y time2 con el método compareTo, devolverá un número mayor o igual que uno si time es mayor que time2.

```
System.out.println("time:" + time);

                  LocalTime time2 = LocalTime.of(10, 12, 20);

                  System.out.println("time2:" + time2);
```

```
                    System.out.println("comparamos time y time2:" +
time.compareTo(time2) );


time:10:19:31.994428400
time2:10:12:20
comparamos time y time2:1
```

Creamos una nueva hora en la variable time3, restando dos horas a time 2 con el método minusHours

```
LocalTime time3 = time2.minusHours(2);

                    System.out.println("restamos dos horas a time2 obteniendo
time3:"+time3);

restamos dos horas a time2 obteniendo time3:08:12:20
```

Calculamos la diferencia entre time2 y time3 que debe ser -120, 2 horas, con la clase MINUTES y el método between.

```
long minutes = MINUTES.between(time2, time3);


                    System.out.println("Diferencia en minutos entre Time2 y
Time3:" + minutes);


Diferencia en minutos entre Time2 y Time3:-120
```

OperarConHoras.java

```
package fechas;

import static java.time.temporal.ChronoUnit.MINUTES;

import java.time.LocalDate;
import java.time.LocalTime;

public class OperarConHoras {



            public static void main(String[] args) {

                    LocalTime time= LocalTime.now();


                    System.out.println("time:" + time);

                    LocalTime time2 = LocalTime.of(10, 12, 20);
```

```
                    System.out.println("time:" + time2);

                    System.out.println("comparamos time y time2:" +
time.compareTo(time2) );


                    LocalTime time3 = time2.minusHours(2);

                    System.out.println("restamos dos horas a time2 obteniendo
time3:"+time3);

                    long minutes = MINUTES.between(time2, time3);


                    System.out.println("Diferencia en minutos entre Time2 y
Time3:" + minutes);



            }

}
```

3.2 La clase LocalDateTime

La clase LocalDateTime es la combinación de las dos anteriores permitiéndonos trabajar con fechas completas, es decir fechas y horas, vamos a poner un breve ejemplo de como crearlas, la manera de operar es igual que en los casos anteriores.

LocalDateTime es un objeto de fecha y hora inmutable que representa una fecha-hora, a menudo vista como año-mes-día-hora-minuto-segundo. También se puede acceder a otros campos de fecha y hora, como día del año, día de la semana y semana del año. El tiempo se representa con una precisión de nanosegundos. Por ejemplo, el valor "2nd October 2007 at 13:45.30.123456789" se puede almacenar en un LocalDateTime.

Obtenemos la fecha y hora actual con el método now en una variable de tipo LocalDateTime.

```
LocalDateTime date= LocalDateTime.now();

System.out.println("valor de fecha actual date" + date);
```

Introducimos fecha con año, mes ,dia , horas (0-23), minutos, segundos con el método of.

```
      LocalDateTime date2 = LocalDateTime.of(2021, 5, 9,21,10,12);
```

Parseamos la fecha con el formato por defecto con el método parse, de años a nano segundos. Entre la fecha y la hora hay que introducir a letra T en el parseo.

```
LocalDateTime date3 = LocalDateTime.parse("2021-08-14T19:34:50.63");

	System.out.println("dia en el año " + date3.getDayOfMonth() + " mes "
+ date3.getMonth() + " Cronología " + date3.getChronology());
```

Podemos crear nuestro propio patrón de parseo con un DateTimeFormatter usando los valores de la tabla anterior.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");

	 // create a LocalDate object and
	LocalDateTime date4
	    = LocalDateTime.parse("06-09-2021 22:10:11", formatter);
```

Y construir patrones más complicados en su caso usando como en el siguiente el día en la semana, la semana en el mes, el mes, el año, y la hora y minutos.

```
  formatter = DateTimeFormatter.ofPattern("ee-W-MM-yyyy H:m");

	 // create a LocalDate object and
	 LocalDateTime dateMes
	    = LocalDateTime.parse("07-2-09-2021 9:34", formatter);
```

ManejoLocalDateTime.java

```
package fechas;

import java.time.LocalDateTime;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class ManejoLocalDateTime {

	public static void main(String[] args) {
	LocalDateTime date= LocalDateTime.now();


	System.out.println("valor de fecha actual date" + date);

	LocalDateTime date2 = LocalDateTime.of(2021, 5, 9,21,10,12);



	System.out.println("dia " + date2.getDayOfMonth() + " mes " +
date2.getMonthValue() + " año " + date2.getYear()
```

```
        + " hora:" + date2.getHour() + " minutos:"+date2.getMinute() + "
segundos" + date2.getSecond());


        LocalDateTime date3 = LocalDateTime.parse("2021-08-14T19:34:50.63");

        System.out.println("dia en el año " + date3.getDayOfMonth() + " mes "
+ date3.getMonth() + " Cronología " + date3.getChronology());


         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");

          // create a LocalDate object and
          LocalDateTime date4
               = LocalDateTime.parse("06-09-2021 22:10:11", formatter);

        System.out.println(date4);

           formatter = DateTimeFormatter.ofPattern("ee-W-MM-yyyy H:m");

          // create a LocalDate object and
           LocalDateTime dateMes
               = LocalDateTime.parse("07-2-09-2021 9:34", formatter);

        System.out.println(dateMes);


          formatter = DateTimeFormatter.ofPattern("DD-yyyy H");

      // create a LocalDate object and
          LocalDateTime date5
           = LocalDateTime.parse("123-2021 21", formatter);


      System.out.println("Fecha con dia en el año y hora:" + date5);



        }

}
```

4 Cadenas en Java. La API String.

Una cadena es una secuencia de caracteres. Las cadenas son una parte fundamental de la mayoría de los programas, así pues Java tiene varias características incorporadas que facilitan la manipulación de cadenas. Java tiene una clase incorporada en el paquete java.lang que encapsula las estructuras de datos de una cadena. Esta clase, llamada String es la representación como objeto de una matriz de caracteres que no se puede cambiar. Hay una clase que la acompaña, llamada StringBuffer, que se utiliza para crear cadenas que pueden ser manipuladas después de ser creadas.

StringFormat es útil para crear cadenas con formatos.

4.1 La Clase String

La clase String nos va a permitir la creación de cadenas pero se recomienda no usar los constructores, sino realizar la asignación directa de literales. Apartir de Java 7 el constructor de String ha sido marcado como deprecated.

```
String cadenavacia="";

String cadena="Cadena con literal";
```

La clase String se almacena internamente como un array unidimensional. Cada carácter en una posición de memoria contigua. Además, el sistema de índices funciona igual que para arrays, el índice inicial es 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C	a	d	e	n	a		c	o	n		l	i	t	e	r	a	l

La Clase String es una clase un poco especial. El asunto con la clase String es que cada vez que asignamos a una variable de tipo cadena se crea una cadena nueva. Con lo que se recomienda cuando ahorrar memoria es importante usar la clase StringBuffer que veremos después para manipulación de cadenas.

Otro detalle importante es que cuando usamos la operación + podemos unir dos cadenas. Cuando ejecutamos la siguiente instrucción se creará un objeto cadena nuevo. Cada vez que realizamos una asignación a un objeto cadena se crea de nuevo.

```
cadena = cadena + " unimos con segunda cadena";

System.out.println(" resultado concatenar " + cadena);
```

La cadena resultado sería
"Cadena con literal unimos con segunda cadena"

Además, cuando usamos el concatenar con otro tipo de datos primitivo, se transforma todo a cadena.

```
double miNum= 5.0;


cadena = cadena + miNum;
```

La cadena resultado sería
`"Cadena con literal unimos con segunda cadena5.0"`

Ejemplo completo `PrimerosStrings.java`

```
package Stringejemplos;

public class PrimerosStrings {

	public static void main(String[] args) {

		String cadenavacia="";

		String cadena="Cadena con literal";

		double miNum= 5.0;


		cadena = cadena + " unimos con segunda cadena";

		System.out.println(" resultado concatenar " + cadena);


		cadena = cadena + miNum;

		System.out.println(" resultado concatenar con double " +
cadena);

	}

}
```

4.1.1 Métodos de comparación de la clase String

En el siguiente enlace tenéis la documentación oficial de Oracle con todos los métodos de la clase String. Vamos a repasar los más relevantes con ejemplos.

https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/String.html

Primero los métodos heredados de Object, **equals**, y como String implementa Comparable implementa **compareTo**:

- **boolean equals(Object anObject)** Nos permite comparar si dos cadenas

de texto son iguales. En el caso de que sean iguales devolverá como valor "true". En caso contrario devolverá "false". Este método tiene en cuenta si los caracteres van en mayúsculas o en minúsculas. Si queremos omitir esta validación tenemos dos opciones. La primera es convertir las cadenas a mayúsculas o minúsculas con los métodos .toUpperCase() y .toLowerCase() respectivamente. Métodos que veremos más adelante. La segunda opción es utilizar el método .equalsIgnoreCase() que omite si el carácter está en mayúsculas o en minúsculas.

- **boolean equalsIgnoreCase(String anotherString)** Compara dos cadenas de caracteres omitiendo si los caracteres están en mayúsculas o en minúsculas.

- **int compareTo(String anotherString)** Este método es un poco más avanzado que el anterior, el cual, solo nos indicaba si las cadenas eran iguales o diferentes En este caso compara a las cadenas léxicamente. Para ello se basa en el valor Unicode de los caracteres. Se devuelve un entero menor de 0 si la cadena sobre la que se parte es léxicamente menor que la cadena pasada como argumento. Si las dos cadenas son iguales léxicamente se devuelve un 0. Si la cadena es mayor que la pasada como argumento se devuelve un número entero positivo. Pero qué es esto de "mayor, menor o igual léxicamente". Para describirlo lo veremos con un pequeño ejemplo.

```
String s1 = "Cuervo";
String s2 = "Cuenca";
System.out.println("Comparacion s1,s2: " + s1.compareTo(s2));
```

Devuelve un número positivo 4 porque la cadena s1 es mayor que la 2. El numero que devuelve indica la posición en carácter del primer carácter que hace a una cadena mayor que otra. El carácter 4 en de s1 es r, de s2 es n. En alfabeto r es mayor que n.

4.1.2 Métodos de búsqueda de caracteres

Tenemos un conjunto de métodos que nos permiten buscar caracteres dentro de cadenas de texto. Y es que no nos debemos de olvidar que la cadena de caracteres no es más que eso: una suma de caracteres.

- **int indexOf(int ch)** Nos devuelve la posición de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista nos devolverá un -1. Si lo encuentra nos devuelve un número entero con la posición que ocupa en la cadena.

- **int indexOf(int ch, int fromIndex)** Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que le indiquemos.

- **int lastIndexOf(int ch)** Nos indica cual es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve un -1.
- **int lastIndexOf(int ch, int fromIndex)** Lo mismo que el anterior, pero a partir de una posición indicada como argumento.

Búsqueda de subcadenas
Este conjunto de métodos son, probablemente, los más utilizados para el manejo de cadenas de caracteres. Ya que nos permiten buscar cadenas dentro de cadenas, así como saber la posición donde se encuentran en la cadena origen para poder acceder a la subcadena. Dentro de este conjunto encontramos:

- **int indexOf(String str)** Busca una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1.

- **int indexOf(String str, int fromIndex)** Misma funcionalidad que indexOf(String str), pero a partir de un índice indicado como argumento del método.

- **int lastIndexOf(String str)** Si la cadena que buscamos se repite varias veces en la cadena origen podemos utilizar este método que nos indicará el índice donde empieza la última repetición de la cadena buscada.

- **lastIndexOf(String str, int fromIndex)** Lo mismo que el anterior, pero a partir de un índice pasado como argumento.

- **boolean startsWith(String prefix)** Probablemente mucha gente se haya encontrado con este problema. El de saber si una cadena de texto empieza con un texto específico. La verdad es que este método podía obviarse y utilizarse el indexOf(), con el cual, en el caso de que nos devolviese un 0, sabríamos que es el inicio de la cadena.

- **boolean startsWith(String prefix, int toffset)** Más elaborado que el anterior, y quizás, y a mi entender con un poco menos de significado que el anterior.

- **boolean endsWith(String suffix)** Y si alguien se ha visto con la necesidad de saber si una cadena empieza por un determinado texto, no va a ser menos el que se haya preguntado si la cadena de texto acaba con otra. De igual manera que sucedía con el método .startsWith() podríamos utilizar una mezcla entre los métodos .indexOf() y .length() para reproducir el comportamiento de .endsWith(). Pero las cosas, cuanto más sencillas, doblemente mejores.

- Boolean contains(String suffix) : devuelve true si la cadena contiene la cadena pasada como parámetro.

Vamos a ver los resultados del siguiente ejemplo:

```
String s1 = "Cuervo";
String s2 = "Cuenca";
```

`s1.charAt(4)` Devuelve v el quinto carácter de s1 Cuervo

`s1.indexOf("C")` Devuelve 0 el primer carácter de s1 Cuervo

`s1.startsWith("C")` **Devuelve true pies s1 empieza por C**

`s2.contains("ue")`); Devuelve true, la cadena contiene s2 Cuenca ue

En el siguiente ejemplo usamos algunos de los métodos. Podéis ver los resultados en la ejecución:

```
Comparacion s1,s2: 4
El caracter en la posición 4 es: v
El indice del caracter 'C' es:0
s1 empieza por el caracter "C":true
s2 contiene la secuencia "ue":true
```

```
package Stringejemplos;

public class StringsMetodos {


	public static void main(String[] args) {

		String s1 = "Cuervo";
		String s2 = "Cuenca";
		int compara= s1.compareTo(s2);
		System.out.println("Comparacion s1,s2: " + compara );

		System.out.println("El caracter en la posición 4 es: " +
s1.charAt(4));


		System.out.println("El indice del caracter \'C\' es:"+
s1.indexOf("C"));

		System.out.println("s1 empieza por el caracter \"C\":" +
s1.startsWith("C"));

		System.out.println("s2 contiene la secuencia \"ue\":" +
s2.contains("ue"));
```

```
	}
}
```

4.1.3 Métodos con subcadenas

Ahora que sabemos como localizar una cadena dentro de otra seguro que nos acucia la necesidad de saber como substraerla de donde está. Si es que no nos podemos estar quietos…

- **String substring(int beginIndex)** Este método nos devolverá la cadena que se encuentra entre el índice pasado como argumento (beginIndex) hasta el final de la cadena origen. Así, si tenemos la siguiente cadena:

```
String s = "Víctor Cuervo";

		System.out.println("subcadena 7 de 'Victor Cuervo': " +
s.substring(7));
```

Substring(7) devuelve Cuervo

String substring(int beginIndex, int endIndex) Si se da el caso que la cadena que queramos recuperar no llega hasta el final de la cadena origen, que será lo normal, podemos utilizar este método indicando el índice inicial y final del cual queremos obtener la cadena. Así, si partimos de la cadena…

```
System.out.println("subcadena (3,8) de 'Victor Cuervo': " + s.substring(3,8));
```

Substring(3,8) devuelve Tor C

4.1.4 Métodos de manejo de caracteres

Otro conjunto de métodos que nos permite jugar con los caracteres de la cadena de texto. Para ponerles en mayúsculas, minúsculas, quitarles los espacios en blanco, reemplazar caracteres,….

String toLowerCase(); Convierte todos los caracteres en minúsculas.

String toUpperCase(); Convierte todos los caracteres a mayúsculas.

```
System.out.println("To UpperCase devuelve:" + s.toUpperCase());
```

String trim(); Elimina los espacios en blanco de la cadena. Importante cuando queremos eliminar espacios al principio y final de la cadena para

comparaciones.

```
String s2= "  Comparaciones elimina espacios    ";

System.out.println("Cadena sin espacios:" + s2.trim());
```

String replace(char oldChar, char newChar) Este método lo utilizaremos cuando lo que queramos hacer sea el remplazar un carácter por otro. Se reemplazarán todos los caracteres encontrados.

```
System.out.println("To replace devuelve:" + s.replace("Cuervo", "Persona"));
```

Ejecución:

```
subcadena 7 de 'Victor Cuervo': Cuervo
subcadena (3,8) de 'Victor Cuervo': tor C
To UpperCase devuelve:VÍCTOR CUERVO
To replace devuelve:Víctor Persona
Cadena sin espacios:Comparaciones elimina espacios
```

Ejemplo completo **StringsMetodosSubcadenas.java** :

```
package Stringejemplos;

public class StringsMetodosSubcadenas {

	public static void main(String[] args) {


		String s = "Víctor Cuervo";

		System.out.println("subcadena 7 de 'Victor Cuervo': " +
s.substring(7));

		System.out.println("subcadena (3,8) de 'Victor Cuervo': " +
s.substring(3,8));


		System.out.println("To UpperCase devuelve:" + s.toUpperCase());



		System.out.println("To replace devuelve:" + s.replace("Cuervo",
"Persona"));

		String s2= "  Comparaciones elimina espacios    ";

		System.out.println("Cadena sin espacios:" + s2.trim());

	}

}
```

4.2 Formato, el método String.format()

El método estático String.format nos va a dar la posibilidad de crear cadenas con formato, usando parámetros.

public static String format(String format, Object... args)

El método format() da formato a una cadena mediante un formato string y argumentos. Por ejemplo, los caracteres 's' y 'S' se evalúan como "null" si el argumento arg es null.

Si arg implementa Formattable, entonces se invoca el método Formattable, entonces se invoca el método arg.formatTo(). De lo contrario, el resultado se evalúa invocando arg.toString().

Para quien haya trabajado con C las cadenas de formato son muy parecidas a las que usamos con el metodo printf y fprintf de C.

Sobrecargas de format

public static String format(String format, Object... args)
public static String format(Locale l, String format, Object... args)

- Donde Locale es un clase Java que nos permite indicar el idioma, el país y la variante del idioma en su caso.
- String format es la cadena con formato a la que se le añadirán los valores de los argumentos, args.

4.2.1 Locale , java.util

Un objeto De configuración regional Locale representa una región geográfica, política o cultural específica. Una operación que requiere una configuración regional para realizar su tarea se denomina sensible a la configuración regional y utiliza la configuración regional para adaptar la información para el usuario. Por ejemplo, mostrar un número es una operación sensible a la configuración regional: el número debe tener un formato de acuerdo con las costumbres y convenciones del país, región o cultura de origen del usuario.

Constructores

Locale(String language): crea una configuración regional a partir de un código

de idioma. Por ejemplo, "fr" es un código de idioma para el francés.

Locale(String language, String country): además del código de idioma y un país, se puede utilizar este constructor con dos parámetros que toman un idioma como primer parámetro y un país como segundo parámetro para crear un objeto de configuración regional.

Locale(String language, String country, String variant): crea una configuración regional a partir del idioma, el país y la variante. Cualquier valor arbitrario se puede utilizar para indicar una variación de una configuración regional.

Ejemplos:

```
Locale locale = new Locale("fr");
    System.out.println("locale: "+locale);

    // Create a locale object using two parameters constructor
    Locale locale2 = new Locale("sp", "ES");
    System.out.println("locale2: "+locale2);

    // Create a locale object using three parameters constructor
    Locale locale3 = new Locale("en", "US", "south");
    System.out.println("locale3: "+locale3);
```

Podemos usar un el método estático Builder que proporciona Java para la creación de objetos Locales.Proporciona métodos como setLanguaje, setRegion, setVariant o setUnicodeLocaleKeyword para cambiar la entrada de teclado.

```
    // A local object from Locale.Builder
    Locale localeFromBuilder = new
Locale.Builder().setLanguage("en").setRegion("GB").build();
    System.out.println("localeFromBuilder: "+localeFromBuilder);
```

4.2.2 Dando formato

La manera de dar formato con String.format es la siguiente. Usamos un modificador especificador de formato %+Carácter. En el siguiente ejemplo indicamos que el argumento es de tipo cadena %s. Lo que hará format es sustituir %S por el valor de la variable nombre y el formato que indiquemos en su caso.

```
String nombre= "Lobito";
float salario = 45.678756885f;

System.out.println(String.format("El nombre es %s",nombre));
```

```
El resultado:

El nombre es Lobito
```

En el segundo ejemplo trabajamos con floats, con punto flotante, usando el modificador %f. Por defector el formato será americano y separá parte entera y decimal por comas.

```
String sf2=String.format("El salario es %f",salario);
              System.out.println(sf2);


El salario es 45,678757
```

En el tercer caso hemos cambiado el formato a Español de España. Además hemos modificado el modificador %f para que nos de como máximo 7 espacios en total y 4 cifras en la parte decimal, `%6.4f,` la precisión.

```
              System.out.println(String.format(locale2,"el valor del salario
es %7.4f",salario));
```

Al ser la región España separará con un punto parte decimal y entera.

```
el valor del salario es 45.6788
```

En el parámetro cadena de formato string format podemos usar diferentes modificadores para indicar el tipo de dato de que usamos en la cadena.

Format Specifier	Data Type	Output
%a	coma flotante (excepto *BigDecimal)*	Devuelve la salida en hexadecimal del número de coma flotante.
%b	Cualquier tipo	"true"si es not null, "false" si el valor es null
%c	character	Carácter Unicode Unicode
%d	integer (incl. byte, short, int, long, bigint)	Decimal Integer
%e	floating point	Número decimal en notación científica

%f	floating point	Numero decimal
%g	floating point	Dependiendo de la precision y el valor el número se ofrecerá en decimal o notación científica
%h	Cualquier tipo	String en hexadecimal del método hashCode() para el tipo recibido
%n	Separador de línea, no recibe ningún tipo	Separador de línea para la plataforma en la que trabajamos
%o	integer (incl. byte, short, int, long, bigint)	Número en Octal
%s	Cualquier tipo	String value
%t	Date/Time (incl. long, Calendar, Date y TemporalAccessor)	%t es el prefijo para las conversiones de fecha/hora. Se necesitan más flags de formato para los tipo Date. Lo explicaremos más adelante
%x	integer (incl. byte, short, int, long, bigint)	Cadena en hexadecimal.

Seguimos algún ejemplo más antes de explicar el formato completo de los modificadores. Podemos pasar mas de un parámetro para el formato. Para cada parámetro añadimos un modificador:

```
	System.out.println(String.format(locale2,"Persona con nombre %s el
valor del salario es %7.4f",nombre, salario));
```

Podemos complicarlo mas todavía haciendo referencia al parámetro que queremos usar en el modificador con $numero, %1$ para el parámetro 1, %2$ para el parámetro 2, etc. Nos permite referenciar el mismo parámetro varias veces. En el siguiente ejemplo referenciamos dos veces salario con %2$

```
		System.out.println(String.format(locale2,"Para el trabajador
%1$20s  con salario %2$7.4f  %n "
				+" o con más precisión es %2$2f y edad %3$f",nombre,
salario, 56.3f));
```

En resumen la sintaxis de modificadores es:

```
%[argument_index$][flags][width][.precision][conversion-character]
```

- **argument_index** es un entero *i* — que indica que la *posición i* del argumento en la lista de argumentos suministrada. Empieza por 1 para referenciar al primer parámetro
- **flags** es un conjunto de caracteres utilizado para modificar el formato de salida.
- **width** es un entero positivo que indica el número mínimo de caracteres que se escribirán en la salida.
- **precision** es un entero que se suele utilizar para restringir el número de caracteres, cuyo comportamiento específico depende de la conversión.
- **conversion-character** es una parte obligatoria que indica qué tipo de datos se reemplaza.

En el ejemplo anterior `%1$20s` indica que la salida tendrá 20 caracteres de tamaño. Se rellena con espacios en caso de que haya menos caracteres

Para numéricos cambia un poco `%2$7.4f` ya que el primer numero 7 indica el número de espacios en total, el segundo la parte decimal.

Los flags

Estos definen formas estándar de modificar la salida y son más comunes para formatear integers y números en coma flotante.

- **Justificar a la izquierda. (Justificar a la derecha es el valor predeterminado).**
- **+ genera un signo más (+) o menos (-) para un valor numérico.**
- **0 obliga a que los valores numéricos sean rellenados a cero a la izquierda. (Los espacios son los predeterminados).**
- **, es el separador de agrupación de comas (para números > 1000).**
- **[space] mostrará un signo menos si el número es negativo o un espacio si es positivo.**

Vamos a mejorar el formato anterior haciendo una tabla. Primero vamos a definir el nombre de las columnas, luego un separador con guiones "-" y luego nuestros datos

```
            System.out.println(String.format(locale2,"%1$-20s %2$-10s %3$-
20s %4$-6s "
                  ,"Nombre"   , "salario","salario-precision", "Edad"));

            System.out.println(String.format(locale2,"%1$-20s %2$-10s %3$-
20s %4$-6s "
                  ,"--------------------"   , "----------","----------------
", "------"));

            System.out.println(String.format(locale2,"%1$-20s %2$-10.4f
%2$-20f %3$-6.1f",nombre, salario, 56.3f));
```

Hemos definido para la primera columna el flag: -20, es decir se alinea a la izquierda y tamaño 20 caracteres. Se rellena con blancos.

Para la segunda -10, 10 caracteres, y en el caso del número le damos precisión cuatro -10.4f.

Para la tercera -20, y para la cuarta -6. En el caso del número de edad, damos -6.1f.

El resultado lo podeis ver tras ejecución:

```
Nombre               salario    salario-precision    Edad
-------------------- ---------- -----------------    ------
Lobito               45.6788    45.678757            56.3
```

4.2.3 Formato para fechas en String.format

Para fechas tenemos nuestros flags en format. Vamos a proporcionar una tabla de flags para fechas y vamos a incluir unos cuantos ejemplos:

Flag	Descripción
%tA	Nombre completo del día de la semana, por ejemplo, "Sunday", "Monday"
%ta	Nombre abreviado del día de la semana, por ejemplo, "Sol", "Mon", etc.
%tB	Nombre completo del mes, por ejemplo, "January", "February", etc.
%tb	Nombre abreviado del mes, por ejemplo, "Jan", "Feb", etc.

%tC	Parte del año del siglo con formato de dos dígitos, por ejemplo, "00" a "99".
%tc	Fecha y hora formateadas con "%ta %tb %td %tT %tZ %tY" por ejemplo "Fri Feb17 07:45:42 PST 2017
%tD	Fecha con formato "%tm/%td/%ty"
%td	Día del mes formateado con dos dígitos. por ejemplo, "01" a "31".
%te	Día del mes formateado sin un 0 principal, por ejemplo, "1" a "31".
%tF	Fecha con formato ISO 8601 con "%tY-%tm-%td".
%tH	Hora del día para el reloj de 24 horas, por ejemplo, "00" a "23".
%th	Igual que %tb.
%tI	Hora del día para el reloj de 12 horas, por ejemplo, "01" – "12".
%tj	Día del año formateado con 0s a la izquierda, por ejemplo, "001" a "366".
%tk	Hora del día para el reloj de 24 horas sin un 0 a la izquierda, por ejemplo, "0" a "23".
%tl	Hora del día para el clic de 12 horas sin un 0 principal, por ejemplo, "1" a "12".
%tM	Minuto dentro de la hora formateado un 0 a la izquierda, por ejemplo, "00" a "59".
%tm	Mes formateado con un 0 a la izquierda, por ejemplo, "01" a "12".
%tN	Nanosegundo formateado con 9 dígitos y llevando 0s, por ejemplo, "000000000" a "999999999".
%tp	Marcador "am" o "pm" específico de la configuración regional.
%tQ	Milisegundos desde la época del 1 de enero de 1970 a las 00:00:00 UTC.
%tR	Tiempo formateado como 24 horas, por ejemplo, "%tH:%tM".

%tr	Tiempo formateado como 12 horas, por ejemplo, "%tI:%tM:%tS %Tp".
%tS	Segundos dentro del minuto formateados con 2 dígitos, por ejemplo, "00" a "60". Se requiere "60" para admitir segundos bisiestos.
%ts	Segundos desde la época 1 de enero de 1970 00:00:00 UTC.
%tT	Tiempo formateado como 24 horas, por ejemplo, "%tH:%tM:%tS".
%tY	Año formateado con 4 dígitos, por ejemplo, "0000" a "9999".
%ty	Año formateado con 2 dígitos, por ejemplo, "00" a "99".
%tZ	Abreviatura de zona horaria. por ejemplo, "UTC", "PST", etc.
%tz	Desvío de zona horaria desde GMT, greenwich, por ejemplo, " -0800 ".

En el siguiente ejemplo vamos a formatear en s fechas completas con diferentes formatos:

El primero va con fecha americana donde el mes va antes del día.

```
        System.out.println("Fecha america:"+ String.format(localeUs ," %tD"
,date ));
```

```
Fecha america: 08/14/21
```

El segundo es un formato construido mas completo con día, abreviatura del mes, año y horas minutos y segundos

```
String fechaCompleta =
                    String.format(localeEs ,"Fecha: %1$td-%1$tb-%1$tY  Hora:
%1$tH:%1$tm:%1$tS" ,date );

        System.out.println("Fecha 24 horas:"+fechaCompleta );
```

```
Fecha 24 horas:Fecha: 14-Aug-2021  Hora: 19:08:50
```

En el tercero haremos que el mes sea numérico y que la hora se de en formato de 12 horas AM y FM.

```
        String fechaCompletaPM =
                        String.format(localeEs ,"Fecha: %1$td-%1$tm-%1$tY  Hora:
%1$tl:%1$tm:%1$tS %1$tp" ,date );

        System.out.println("Fecha AMPM:"+fechaCompletaPM );
```

```
Fecha AMPM:Fecha: 14-08-2021  Hora: 7:08:50 pm
```

StringFormatFechas.java

```
package formato;

import java.time.LocalDateTime;
import java.util.Locale;

public class StringFormatFechas {

       public static void main(String[] args) {


             LocalDateTime date = LocalDateTime.parse("2021-08-
14T19:34:50.63");

          Locale localeUs = new Locale("en", "US");
        System.out.println("locale: "+localeUs);



        // Create a locale object using two parameters constructor
        Locale localeEs = new Locale("sp", "ES");
        System.out.println("locale2: "+localeEs);


        System.out.println("Fecha america:"+ String.format(localeUs ," %tD"
,date ));

        String fechaCompleta =
                        String.format(localeEs ,"Fecha: %1$td-%1$tb-%1$tY  Hora:
%1$tH:%1$tm:%1$tS" ,date );

        System.out.println("Fecha 24 horas:"+fechaCompleta );

        String fechaCompletaPM =
```

```
                    String.format(localeEs ,"Fecha: %1$td-%1$tm-%1$tY  Hora:
%1$tl:%1$tm:%1$tS %1$tp" ,date );

        System.out.println("Fecha AMPM:"+fechaCompletaPM );

      }

}
```

4.3 La clase StringBuffer

Un objeto String representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada. Un objeto StringBuffer representa una cadena cuyo tamaño puede variar.

La clase StringBuffer dispone de muchos métodos para modificar el contenido de los objetos StringBuffer. Si el contenido de una cadena va a ser modificado en un programa, habrá que sacrificar el uso de objetos String en beneficio de StringBuffer, que aunque consumen más recursos del sistema, permiten ese tipo de manipulaciones.

La gran ventaja de StringBuffer es que es Thread.safety. Cuando muchos hilos acceden a la vez a una variable compartida de tipo StringBuffer, la exclusión mutua se realiza correctamente.

Documentación oficial de Oracle

https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/StringBuffer.html

En las siguientes tablas presentamos los constructores y los métodos para StringBuffer.

Método	Función
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Empleado para añadir al objeto un String (String) o una variable (int, char, double, ...)
capacity()	Devuelve el espacio libre del StringBuffer.
charAt(int)	Devuelve el caracter de la posición especificada (int)
getChars(int, int, char[], int) Copia los caracteres indicados en la posiciÃ³n indicada de un string	
insert(int, ...)	Inserta en la posición indicada (int) de un StringBuffer, un String(String) o un valor (int, double, float, ...)

length()	Devuelve el número de ccaracteres del String
reverse()	Cambia el orden de los caracteres de un String
setCharAt(int, char)	Cambia el caracter en la posiciÃ³n indicada
setLength(int)	Cambia el tamaño (int) de un StringBuffer
toString()	Convierte el objeto en un String

En el siguiente ejemplo veremos el uso de unos cuantos de estos métodos.

Con el constructor new StringBuffer(20); aseguramos que la capacidad inicial es 20. El resto de métodos están bastante claros en su ejecución.

A destacar insert que inserta un valor en la posición que le dices en la cadena.

```
Uno.insert(10, "-Algo entre medias-");
```

```
CadenaHola Mundo-Algo entre medias-Adios Mundo
```

EjemploStringBuffer

```
package stringbuffer;

public class EjemploStringBuffer {


	public static void main(String[] args) {


	    StringBuffer Dos = new StringBuffer( 20 );
	    StringBuffer Uno = new StringBuffer( "Hola Mundo" );
	    Dos.append("casa al a");

	    Uno.append("Adios Mundo");

	    System.out.println("Capacidad:" + Uno.capacity());
	    Uno.ensureCapacity(100);
	    System.out.println("Capacidad:" + Uno.capacity());
	    Uno.insert(10, "-Algo entre medias-");
	    System.out.println("Cadena" + Uno.toString());
	    System.out.println("Longitud:" + Uno.length());
	    System.out.println("Cadena Dos:"+Dos);
	    System.out.println("REverse:" + Dos.reverse());

	}

}
```

4.4 StringBuilder

Ofrece las mismas funcionalidades que StringBuffer, es como StringBuffer, pero no es thread Safety, detalle a tener en cuenta. En cambio, es más rápida, se recomienda para programas en los que se trabaja con un solo hilo.

Constructor	Descripción	Ejemplo
StringBuilder()	Construye un StringBuilder vacío y con una capacidad por defecto de 16 carácteres.	StringBuilder s = new StringBuilder();
StringBuilder(int capacidad)	Se le pasa la capacidad (número de caracteres) como argumento.	StringBuilder s = new StringBuilder(55);
StringBuilder(String str)	Construye un StringBuilder en base al String que se le pasa como argumento.	StringBuilder s = new StringBuilder("hola");

Retorno	Método	Explicación
StringBuilder	append(...)	Añade al final del StringBuilder a la que se aplica, un String o la representación en forma de String de un dato asociado a una variable primitiva
int	capacity()	Devuelve la capacidad del StringBuilder
int	length()	Devuelve el número de caracteres del StringBuilder
StringBuilder	reverse()	Invierte el orden de los caracteres del StringBuilder
void	setCharAt(int indice,char ch)	Cambia el carácter indicado en el primer argumento por el carácter que se le pasa en el segundo
char	charAt(int indice)	Devuelve el carácter asociado a la posición que se le indica en el argumento
void	setLength(int	Modifica la longitud. La nueva longitud

	nuevaLongitud)	no puede ser menor
String	toString()	Convierte un StringBuilder en un String
StringBuilder	insert(int indiceIni,String cadena)	Añade la cadena del segundo argumento a partir de la posición indicada en el primero
StringBuilder	delete(int indiceIni,int indiceFin)	Borra la cadena de caracteres incluidos entre los dos indices indicados en los argumentos
StringBuilder	deleteChar(int indice)	Borra el carácter indicado en el índice
StringBuilder	replace(int indiceIni, int indiceFin,String str)	Reemplaza los caracteres comprendidos entre los dos indices por la cadena que se le pasa en el argumento
int	indexOf (String str)	Analiza los caracteres de la cadena y encuentra el primer índice que coincide con el valor deseado
String	subString(int indiceIni,int indiceFin)	Devuelve una cadena comprendida entre los dos índices

4.4.1 Tarea StringBuilder

Transformar el ejemplo de StringBuffer a StringBuilder, y hacer que funcione de la misma manera

4.4.2 La clase formatter

Nos permite realizar exactamente lo mismo que String.format pero además podemos usar StringBuilder en la combinación.

Funciona como intérprete para cadenas de formato de estilo printf. Esta clase proporciona compatibilidad con la justificación y alineación del diseño, formatos comunes para datos numéricos, de cadena y de fecha y hora, y salida específica de la configuración regional. Se admiten tipos de Java comunes como byte, BigDecimal y Calendar. La personalización de formato limitado para tipos de usuario arbitrarios se proporciona a través de la interfaz Formattable.

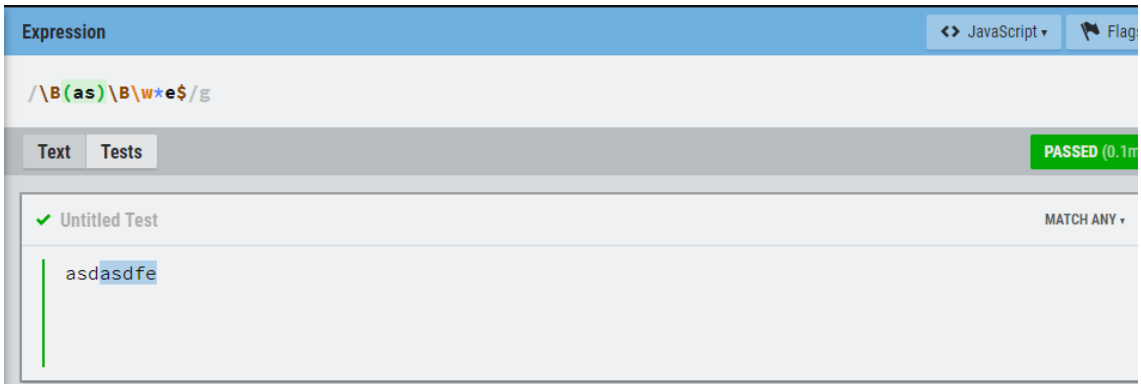
5 Expresiones Regulares y programación Java

5.1 Expresiones Regulares

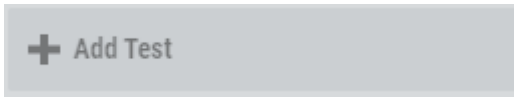
En estos apuntes trataremos con las expresiones regulares. Son utilizadas en muchos lenguajes de programación como Java, c#, y JavaScript, o XSL, XML Schema, XPATH, árbol DOM, etc. Nos sirven para crear patrones de búsqueda o validación. También es usado por el Shell de Linux y UNIX para búsqueda de ficheros o texto en ficheros.
Por ejemplo, quiero que mi campo teléfono tenga 9 dígitos. Para ello escribimos una expresión regular del tipo [0-9]{9}. Aquí estoy indicando que mi campo teléfono contenga dígitos del 0 al 9 ([0-9], 9 veces ({9}).

5.2 Herramientas para probar las expresiones regulares

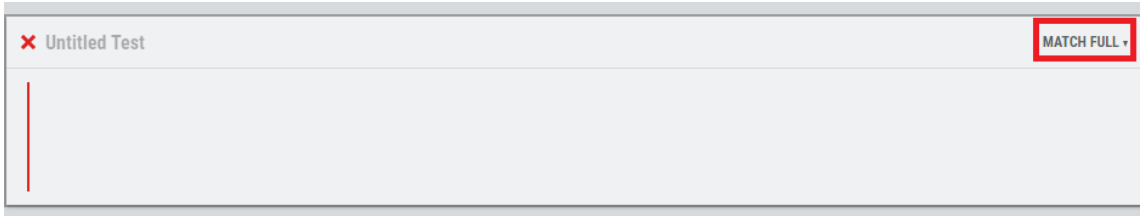
Usar la página https://regexr.com/. En esta os encontrareis una página principal en la parte superior la zona de expresiones regulares. En la parte inferior introduciréis un texto de prueba en la pestaña text.



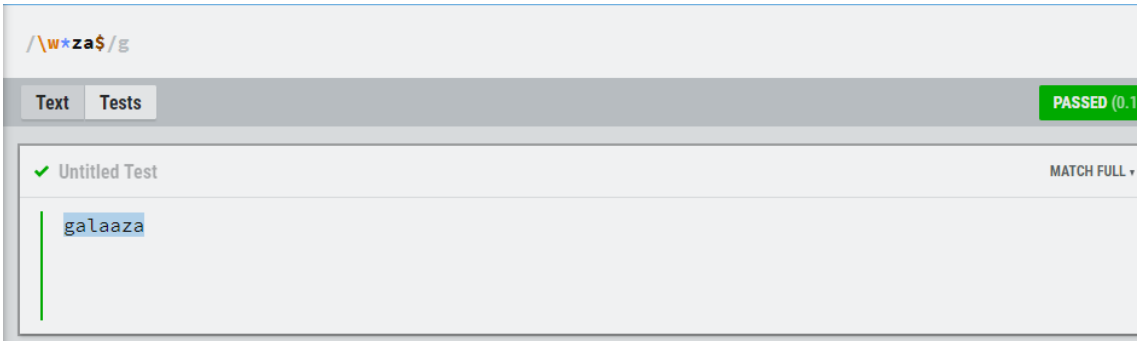
Si no aparece tests, pulsar en añadir uno.



Recordar elegir match full, cuando la expresión tenga que validar una palabra completa no buscar palabras o partes dentro de una cadena, la mayoría de los casos en los ejercicios.



Por ejemplo, crear una expresión regular que valida palabras que terminen por "za" se selecciona entera, con match full:



5.2.1 Conceptos Básicos

En esta tabla puede encontrar las convenciones y los caracteres más importantes, así como sus significados. Los 11 caracteres [ ] ( ) { } | ? + - * ^ $ \ y . son metacaracteres y tienen un significado especial dentro de las expresiones regulares, que se explican en las siguientes secciones. Si desea utilizar uno de estos caracteres como este carácter en una expresión regular, puede usar un \ delante del carácter para omitir el significado como carácter meta. Todos los demás caracteres pueden usarse en una expresión regular como tal.

Expresión Regular	Significado y Ejemplo
a	La expresión regular "a" también coincide con "a". Siempre que el carácter no sea un metacaracteres con otro significado, se puede usar directamente en la expresión regular. *Ejemplo: **a**bc defg **a**bcdefgb**a**fcgbde 0123456789*
[abc]	Los corchetes [y] se pueden utilizar para definir un grupo de caracteres. El ejemplo encuentra uno de los caracteres a, b o c. En resumen, que contenga a, b o c. *Ejemplo: **abc** defg **abc**defgb**a**f**c**g**b**de 0123456789*
[a-f]	El guión - se puede utilizar para definir un rango de caracteres. El ejemplo encuentra uno de los caracteres a, b, c, d, e o f. También en este ejemplo, la expresión regular solo coincide con un carácter. *Ejemplo: **abc def**g **abcdef**g**bafc**g**bde** 0123456789*
[0-9]	El guión también se puede utilizar para definir un rango de números. El ejemplo encuentra los números del 0 al 9. *Ejemplo: abc defg abcdefgbafcgbde **0123456789***

[A-Z0-9abc]	Dentro de un grupo de caracteres, se pueden usar rangos y caracteres únicos. En el ejemplo, el grupo consta de las letras mayúsculas de la A a la Z, los dígitos del 0 al 9 y las letras minúsculas a, b y c. *Ejemplo:* ***abc defg abcdefgbafcgbde 0123456789*** *-&*
[^a]	Con el carácter meta ^ al principio de un grupo de caracteres, el grupo de caracteres se niega. Eso significa que, el ejemplo coincidirá con cualquier carácter pero no con un a. *Ejemplo: a****bc defg*** *a****bcdefgb****afcgbde 0123456789***
^a	Si el metacarácter ^ no se incluye en un grupo de caracteres, significa el comienzo de una cadena o una línea. El ejemplo coincidiría con todas las líneas o cadenas que comienzan con a. *Ejemplo:* ***a****bc defg abcdefgbafcgbde 0123456789*
a$	Así como el metacarácter ^ representa el comienzo de una cadena o línea, el carácter $ representa su final. Así que todas las cadenas que terminan con una a se encontrarían. *Ejemplo: abc defg abcdefgbafcgbde 0123456789****a***
^abc$	Aquí los metacaracteres ^ y $ se usan juntos. Este ejemplo coincidiría con todas las cadenas o líneas que son iguales a "abc". *Ejemplo 1:* ***abc.*** *Ejemplo 2: abc abc.*
\b	Representa la posición al principio o al final de una palabra. Se usa para los límites de la palabra, principio o fin
\B	Representa una posición que no está al principio o al final de una palabra. Se usa sólo para dentro de la palabra
\babc\b	Encuentra la única palabra "abc", pero no la cadena "abc", cuando está rodeada por otros caracteres. *Ejemplo:* ***abc*** *abcde* ***abc*** *deabcde deabc* ***abc***
\babc	Encuentra todas las palabras que comienzan con "abc". *Ejemplo:* ***abc abc****de* ***abc*** *deabcde deabc* ***abc***
abc\b	Encuentra todas las palabras que terminan con "abc". *Ejemplo: abc abcde abc deabcde de****abc*** *abc*
\Babc\B	Busca todas las palabras que contienen "abc", pero que no comienzan ni terminan con "abc". *Ejemplo: abc abcde abc de****abc****de deabc abc*
abc\B	Busca todas las palabras que contienen "abc", pero no termina con "abc".

	*Ejemplo: abc **abc**de abc de**abc**de deabc abc*
ABC|abc	El carácter | representa una alternativa. Este regex encontrará "ABC" y "abc". *Ejemplo: **abc**de **ABC**DE fg**abc**de FG**ABC**DE*
[a\-f]	Si hay un \ delante de un meta carácter, se omite el significado de este meta carácter, y se toma como un carácter normal "-". En este caso, el metacarácter no representa un rango, pero se utiliza como propio carácter. Entonces, el ejemplo coincide con los caracteres a, f y -. En otras palabras, con el carácter \ es posible agregar metacaracteres a los grupos de caracteres. *Ejemplo: **a**bcde**f**g **a**bcde**f**gh **-***
1\+1=2	Esta expresión regular coincide con la cadena 1+1=2. De nuevo, el carácter meta + se omite con \. Y funciona como una carácter "+" normal. *Ejemplo: **1+1=2** 1\+1=2*
[-af]	Cuando un metacarácter se encuentra dentro de un grupo de caracteres en una posición sin significado, se utiliza como carácter normal. El ejemplo coincide con los caracteres -, a y f. *Ejemplo: **a**bcde**f**g **a**bcde**f**gh **-***
[af-]	Lo mismo se aplica a - al final de un grupo. *Ejemplo: **a**bcde**f**g **a**bcde**f**gh **-***
ab[cd]	En este ejemplo, un carácter de un grupo de caracteres se combina con "ab". Entonces, el ejemplo coincide con las cadenas "abc" y "abd". *Ejemplo: **abc**def **abd**ef cd**abc**d cd**abd**c*
.	El punto representa un carácter arbitrario. Cada carácter será encontrado con esta expresión regular. Depende del modificador, si se incluyen los saltos de línea. *Ejemplo: **abc defg abcdefgbafcgbde 0123456789 -&***
.ab	Coincide con cualquier cadena con tres caracteres de los cuales los dos últimos caracteres son a y b, por ejemplo aab, eab, %ab, :ab, etc. *Ejemplo: abc df**gab**c **&ab** fg**xab***
[^a]ab	Encuentra las mismas cadenas que la expresión regular ".ab" excepto la cadena "aab". *Ejemplo: aab **cab** dd **dab** fg **&ab***

5.2.2 Repeticiones

Para describir situaciones en las que hay repeticiones de caracteres o clases de

caracteres completos, puede utilizar algunos de los siguientes metacaracteres.

Expresión Regular	Significado y Ejemplo
ab{2}	El elemento anterior debe aparecer exactamente dos veces. En este ejemplo, esto solo corresponde a abb. La a no se agrega con un gancho a una clase con la b, por lo que la expresión repetida solo afecta a la b y no a la a. *Ejemplo: a ab **abb** b**abb** **abb**b **abb**bb b**abb**bbbbbbbbd*
ab{2,3}	El elemento anterior debe aparecer al menos dos veces y como máximo tres veces. Esta expresión regular encontraría abb, abbb, pero no ab o abbbb. *Ejemplo: a ab **abb** babb **abbb** **abbb**b b**abbb**bbbbbbbd*
ab{2,}	El elemento anterior debe aparecer al menos dos veces. El ejemplo corresponde a abb, abbb, abbbb y así sucesivamente. *Ejemplo: a ab **abb** b**abb** **abbb** **abbbb** b**abbbbbbbbbb**d*
ab{,3}	El elemento anterior no debe aparecer más de tres veces. El ejemplo corresponde a abbb y abbbb pero no a abbbbb. *Ejemplo: a ab **abb** babb **abb**b **abb**bb b**abb**bbbbbbbbd*
ab?	El signo de interrogación indica que el carácter anterior es opcional. Esto significa que el elemento anterior puede aparecer, pero no es necesario que aparezca. El ejemplo encontraría a y ab. El signo de interrogación corresponde a la expresión {0,1}. *Ejemplo: **a ab** **ab**b b**ab**b **ab**bb **ab**bbb b**ab**bbbbbbbbbd*
ab+	El signo más indica que el elemento anterior debe aparecer al menos una vez, pero también varias veces. El ejemplo correspondería a ab, abb, abbb y así, pero no a. El signo más es lo mismo que la expresión {1,}. *Ejemplo: a **ab** **abb** b**abb** **abbb** **abbbb** b**abbbbbbbbbb**d*
ab*	El carácter * significa que el carácter anterior debe aparecer cero o más veces. Es lo mismo que la expresión {0,}. *Ejemplo: **a** **ab** **abb** b**abb** **abbb** **abbbb** b**abbbbbbbbbb**d*
[ab]+	Este ejemplo encuentra cadenas como a, b, ab, ba, abb, ababa así sucesivamente. Esto no significa que se deba repetir el mismo carácter anterior. Esto significa que los caracteres del grupo deben ser repetidos. Si desea encontrar repeticiones del mismo carácter, debe usar referencias anteriores. La expresión regular sería ([ab])\1+ y se explicará a continuación. *Ejemplo: **a** **ab** **abb** **babb** **abbb** **abbbb** **babbbbbbbbbb**d*
a[bc]+d	Esta expresión encuentra cadenas como abd, acd, abcd, acbd, abccbd, acbcbcbd así sucesivamente. *Ejemplo: **abcd** aad **acbd** fg **abbccbbcd** fg **abd** fg **acd***
[0-9]{2,3}	También en este ejemplo, no es necesario que se repitan

	los mismos dígitos. La expresión corresponde a todos los números con dos o tres dígitos, por lo que los números del 10 al 999. Cadenas como 1,2 no serán encontradas por esta expresión. *Ejemplo: 1,4 **10 89** ab3a ab**42**a **23**4*
[0-9,A-F]{4}	Número hexadecimal de cuatro cifras **FA45** FA3

5.2.3 Clases de Carácteres

Detrás de la creación de grupos de caracteres propios entre corchetes, también hay algunas clases de caracteres predefinidas. Con estas clases, las expresiones regulares se vuelven más cortas y claras.

Expresión Regular	Significado y Ejemplo
\d	Esta expresión representa un dígito, por lo que es equivalente a [0-9]. *Ejemplo: abc defg abcdefgbafcgbde **0123456789** -&*
\D	Esta expresión representa un carácter que no es un número. Esto sería [^0-9] o [^\d]. *Ejemplo: **abc defg abcdefgbafcgbde** 0123456789 **-&***
\w	Esta expresión (word) representa una letra, un número o un guión bajo. Por lo tanto, \w coincide con la clase de caracteres [A-Za-z0-9_]. *Ejemplo: **abc defg abcdefgbafcgbde 0123456789** -&*
\W	Esta expresión representa cualquier carácter que no sea un número, una letra o un guión bajo. Es lo mismo que [^\w] o [^A-Za-z0-9_]. *Ejemplo: abc defg abcdefgbafcgbde 0123456789 **-&***
\s	Esta expresión corresponde a whitespace, por ejemplo, un salto de línea, una pestaña, un espacio, etc.
\S	Esta expresión representa cualquier carácter que no sea whitespace, es lo mismo que [^s].

5.2.4 Agrupación y Referencias Posteriores

Con los paréntesis (y), puede agrupar algunos caracteres, por ejemplo, para aplicar un operador a todo el grupo. Además, con los paréntesis puede crear referencias posteriores. Eso significa que los caracteres que se encuentran en este paréntesis se almacenarán, de modo que pueda reutilizarlos en la misma

expresión regular o incluso en la expresión regular de reemplazo, cuando busque y reemplace con expresiones regulares.

Expresión Regular	Significado y Ejemplo
(ab)+	El grupo entero "ab" se repite una o más repeticiones de "ab" será encontrado. *Ejemplo: **ab**cde **abab**cde **abababab**a*
ab(cd|ef|gh)i	Entre los paréntesis, hay algunas alternativas. Este ejemplo coincide con las cadenas "abcdi", "abefi" y "abghi", pero no otras cadenas. *Ejemplo: **abcdi** abcdefghi **abghi***
(ab){2}	El grupo ab se repit dos veces. ab ab **abab abab**ab
(\d+\.)(\d+\.)	Las referencias no solo se pueden utilizar en una sola expresión regular.
\ba\b\s\b([aoeiu][a-z]+)\b	Con esta expresión regular, puede encontrar todas las palabras simples "a" seguidas por otra palabra que comienza con a, o, e, i o u. En inglés, no está permitido escribir una "a" delante de una palabra que comienza con una vocal.

5.2.5 Modificadores

El comportamiento de las expresiones regulares puede ser modificado por modificadores. Si desea modificar estos modificadores :

- **Modificador i (Case Insensitive - no distingue mayúsculas y minúsculas):** Si este modificador está habilitado, la búsqueda no distingue entre mayúsculas y minúsculas. Esto significa que la expresión regular [az] coincide solo con letras minúsculas (solo "a" a "z" - el modificador i no está activo) o todos los caracteres en minúscula y mayúscula ("a" a "z" así como de "A" a "Z" - el modificador i está activo).
- **Modificador m (Multi Line - multilínea):** Si este modificador está activo, el texto completo se tratará como varias líneas. Esto significa que ^ y $ corresponden al principio y al final de todo el archivo. Si el modificador no está activo, ^ y $ coincidirán con el principio y el final de una línea.
- **Modificador s (Single Line - línea única):** Procesa una cadena como una sola línea. Si este modificador está activo, el punto . coincide con todos los caracteres, incluidos los espacios. Si este modificador no está activo, este no será el caso.
- **Modificador g (Greedy Mode):** Este modificador modifica todos los operadores siguientes, como + y *. Si este modificador está activo, todos

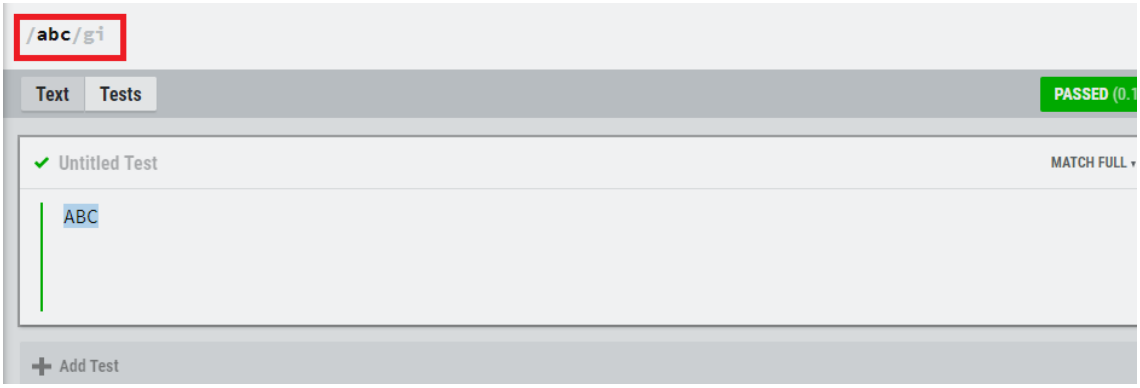
los operadores se comportarán normalmente. Si este modificador no está activo, las expresiones regulares se aplicarán non greedy. Eso significa que el + funciona como +?, el * funciona como *? y así enseguida.

- **Modificador x (Extended Syntax - sintaxis extendida):** Si este modificador está activo, puede usar whitespace (como espacios) en sus expresiones regulares y agregar comentarios (después de # en una línea, todos los demás caracteres no se usarán para la expresión regular). Con esto, la expresión regular será más legible, pero tendrá que escapar de todos los espacios con \ siempre que no se use en un grupo de caracteres.

Si un modificador solo se debe usar para una expresión regular única o para una parte de una expresión regular, puede usar los siguientes métodos para modificar los modificadores. Los modificadores mencionados anteriormente se nombran por sus letras, lo que significa que deben usarse las letras i, m, s, g o x.

Expresión Regular	Significado y Ejemplo
(?i)[abc] o abc/i	En este ejemplo, puede ver cómo activar un modificador. En este ejemplo, el modificador i para la insensibilidad a los casos de mayúsculas y minúsculas. *Ejemplo:* ***abc****def* ***ABC****DEF*
(?i)[a](?-i)[cd]	Al usar (?-I), se deshabilita un modificador. En el ejemplo, el modificador i se activa primero, se encontrarán las letras a y A. Después de deshabilitar el modificador i, c y d deben estar en minúsculas para coincidir con esta expresión. *Ejemplo:* ***ac Ac*** *AC* ***ad Ad*** *AD*
((?i)[a])[cd]	Con paréntesis, puede lograr los mismos resultados. Por supuesto, i debe ser discapacitado en general en este ejemplo. *Ejemplo:* ***ac Ac*** *AC* ***ad Ad*** *AD*
(?ig-msx)[abc]	Si desea modificar varios modificadores al mismo tiempo, también puede hacerlo en una sola expresión. Otras posibilidades son (?ims) para activar algunos modificadores o (?-ims) para deshabilitar varios modificadores.

El validador que usamos los acepta sólo el segundo formato, y siempre a final de linea. Dejar /g a no ser que os indique lo contrario. Os los indico porque se pueden usar luego en otros lenguajes, pero para XML no lo vamos a usar.



5.3 Practica guiada de expresiones regulares

Validar los siguientes ejemplos en página https://regexr.com/:

1. Expresión regular para un entero

```
"^-?[0-9]+$"
```

4,-5

2. Expresión regular para un entero positivo

```
"^[0-9]+$"
```

5, 6

3. Podéis hacer la expresión para un entero negativo.

4. Validando DNI

```
^[0-9]{8}
Ej:44555666
```

5. Expresión regular para un teléfono de 9 dígitos, como 91234556.

```
^[0-9]{8}
```

6. Cádenas que empiecen por ba, como baya.

```
^(ba)\w*
```

7. Cádenas que empiecen por y terminen por es, bananes.

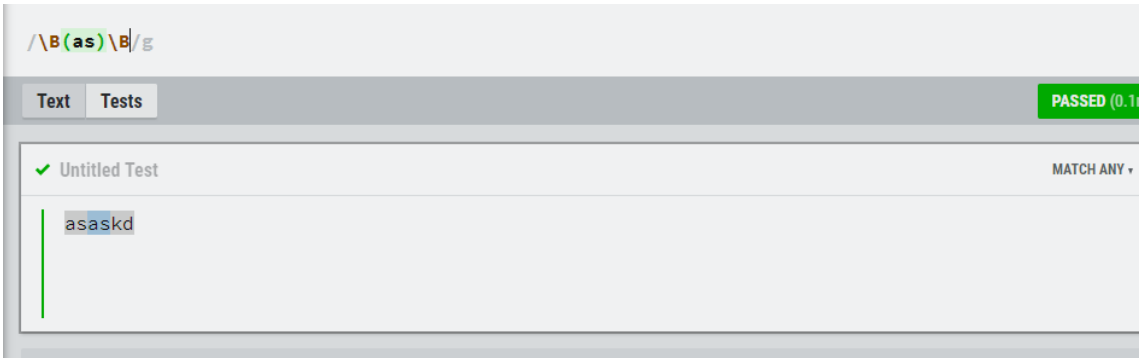
```
^(ba)\w*(es)
```

8. Cádenas que empiecen por a, contengan un . y acaben por s como ahora.sis.

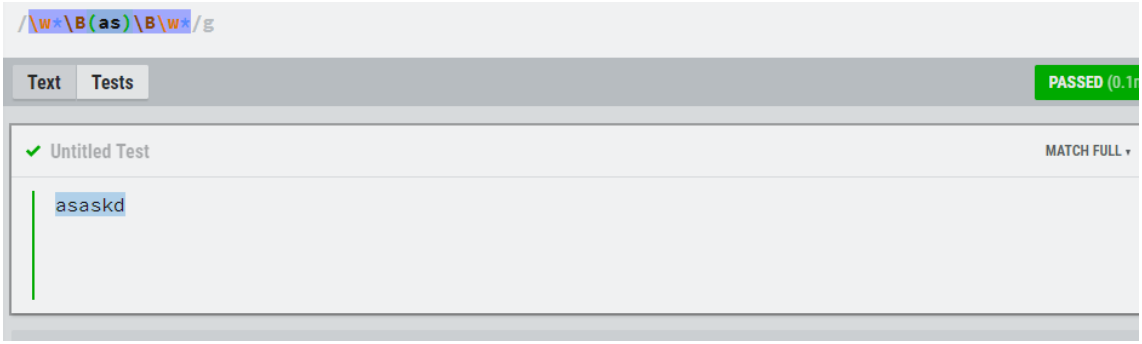
```
^a\w*(\.)\b\w*s\b  ó
^a\w*(\.)\b\w*s$
```

9. Cádenas que contengan “as”, pero ni empiecen ni terminen por as. Por ejemplo hojarasca. S

```
\B(as)\B hace un match parcial con ANY
```




```
\w*\B(as)\B\w* hace match total
```



10. Expresión que valida nombre de ficheros con extensiones de 3 caracteres, como por ejemplo, ejemplo.zip. El tamaño máximo del nombre de fichero es 255, el mínimo 1. El nombre puede contener letras, digitos y guión bajo. Las extensiones sólo contienen letras y dígitos.

```
^\w{1,255}\.[A-Za-z]{3}
```

11. Palabra que empiece por un numero entre 1 y 5 , y una letra entre A y D

```
^[1-5,A-D]\w*
```

5.4 Practica independiente

Nota : Os indico donde debéis usar match any en el test en los ejercicios.

Ejercicio 1. Podéis añadirle la letra, para que sea el NIF, a la validación. Las letras que admite el DNI son TRWAGMYFPDXBNJZSQVHLCKE.

Ejercicio 2. Validación de números binarios. Ejemplo 11001

Ejercicio 3. Podéis validar un número octal. Ejemplo 2345

Ejercicio 4. Validar número hexadecimal. Por ejemplo, xF03 (Lo empezamos por x)

Ejercicio 5. Validar un número real con 4 decimales.

Ejercicio 6. Validar fecha numérica en formato dd/mm/aa

Por ejemplo 21/11/2001

Ejercicio 7. Valida un nombre de usuario en twitter, empieza por @ y puede contener letras mayúsculas y minúsculas, números, guiones y guiones bajos.

Ejercicio 8. Validar ISBN de 13 digitos, siempre empieza en 978- o 979- .

Ejercicio 9. Validar número de teléfono con prefijo + dos digitos y teléfono de 9 digitos

Ejemplo +34999889898

Ejercicio 10. Validar un código de pedido que siempre empieza por P mayúscula contiene 6 caracteres alfanuméricos y termina por EMPR.

Ejercicio 11. Coincidencia con cualquier dirección de correo electrónico de los dominios yahoo.com, hotmail.com y gmail.com. match any)

Ejercicio 12. Buscar una o más letras sueltas separadas por espacios. Por ejemplo, "a c é" es válida, pero "a c de" no. (match any)

Ejercicio 13. dos o más letras sueltas separadas por espacios. Por ejemplo, "a c é" es válida, pero "d" no. match any)

```
(\d(\s)+){2,}
```

Ejercicio 14. Buscar una o más palabras (sólo letras inglesas minúsculas, separadas por uno o varios espacios), usar modificador /i en vez de /g (match any).

Ejercicio 15. Validar una única palabra en mayúsculas.

Ejercicio 16. Validar un único número sin signo y con como mucho dos decimales (el separador puede ser punto o coma, pero sólo puede estar si hay decimales).

Ejercicio 17. Validar un único número con signo (más o menos) y con decimales (el separador puede ser punto o coma, pero sólo puede estar si hay decimales).

Ejercicio 18. Validar contraseña (al menos seis caracteres, puede contener letras, números y los caracteres * + . - _, pero no espacios u otros caracteres).

Ejercicio 19. Explicar la siguiente validación de correo electrónico con vuestras palabras.

```
^([\\w-]+\\.)*?[\\w-]+@[\\w-]+\\.([\\w-
]+\\.)*?[\\w]+$"
```

Ejercicio 20. Construir una expresión regular para validar que empiecen por “es”, contengan un guion o un $ y terminen por y.

6 Expresiones Regulares en Java

7 El paquete `java.uti.regex`

La API de Java contiene, en el paquete , una serie de clases que nos permiten determinar si una secuencia de caracteres se ajusta a un patrón definido por una expresión regular. Esas clases se muestran en el diagrama de clases de la figura 11.1.

7.1 La Interfaz MatchResult

Interfaz MatchResult declara métodos que nos permiten consultar el resultado de ajustar una secuencia de caracteres contra una expresión regular. Los métodos de esta interfaz se muestran en la figura 11.8.

Tabla Métodos de la Interfaz MatchResult

int start() Regresa la posición, dentro de la secuencia de caracteres, del primer carácter de una coincidencia. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló.

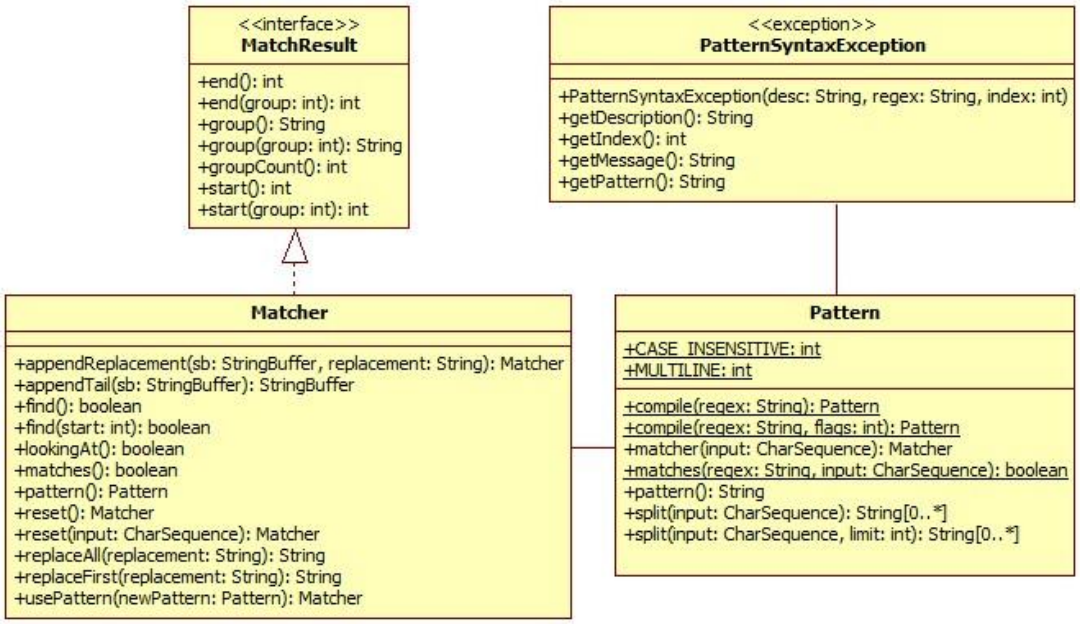


Figura 11.1 Diagrama de Clases de las Clases de Expresiones Regulares

Tabla 11.8 Métodos de la Interfaz MatchResult. Cont.

int start(int group) Devuelve la posición, dentro de la secuencia de caracteres capturada por el grupo del parámetro, del primer carácter de una coincidencia. Si hubo una coincidencia pero no con el grupo mismo, el método regresa -1. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló. IndexOutOfBoundsException – Si no hay un grupo capturado en el patrón con el índice del parámetro.

int **end**()
Devuelve la posición, dentro de la secuencia de caracteres, después del último carácter de una coincidencia. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló.

int **end**(int group)
Devuelve la posición, dentro de la secuencia de caracteres capturada por el grupo del parámetro, después del último carácter de una coincidencia. Si hubo una coincidencia pero no con el grupo mismo, el método regresa -1. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló. IndexOutOfBoundsException – Si no hay un grupo capturado en el patrón con el índice del parámetro.
String **group**() Devuelve la subsecuencia de la secuencia de caracteres que se ajusta al patrón. Si el parámetro se ajusta a una cadena vacía, el método regresa una cadena vacía. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló.
String **group**(int group) Devuelve la subsecuencia de la secuencia de caracteres capturada por el grupo del parámetro. Si hubo una coincidencia pero no con el grupo mismo, el método regresa null. Si el parámetro se ajusta a una cadena vacía, el método regresa una cadena vacía. Lanza: IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló. IndexOutOfBoundsException – Si no hay un grupo capturado en el patrón con el índice del parámetro.
int **groupCount**() Devuelve el número de grupos en el patrón. No incluye el grupo cero.

7.2 La Clase Matcher

La clase Matcher es la clase que realiza las operaciones de ajuste de una secuencia de caracteres a un patrón definido por la clase Pattern. Un objeto Matcher, que es una instancia de la clase Matcher se crean invocando al método matcher() de un patrón, que es una instancia de la clase Pattern. Algunos de los métodos de la clase Matcher y que podemos usar para realizar tres diferentes tipos de operaciones de ajuste, se muestran en la tabla siguiente:

Tabla Métodos de la Clase Matcher.

public boolean **matches**() Intenta ajustar la secuencia de caracteres completa al patrón. Regresa true si y sólo sí la secuencia completa se ajusta al patrón. Si el ajuste tiene éxito, se puede obtener más información usando los métodos: start(), end(), y group().
public boolean **find**() Intenta encontrar la siguiente subsecuencia de la secuencia de caracteres que se ajusta al patrón. Devuelve true si y sólo sí hay una subsecuencia de la secuencia de caracteres que se ajusta al patrón. Este método inicia al principio de la secuencia o si la invocación previa tuvo éxito, y el ajustador no fue restaurado, al primer carácter que no coincidió en el ajuste previo. Si el ajuste tiene éxito, se puede obtener más información usando los métodos: start(), end(), y group().
public boolean **find**(int start) Restablece este matcher y luego intenta encontrar la siguiente subsecuencia de la secuencia de caracteres que se ajusta al patrón, a partir del índice especificado por el parámetro start. Devuelve true si y sólo sí hay una subsecuencia de la secuencia de caracteres que se ajusta al patrón, a partir del índice especificado. Las siguientes invocaciones del método find() iniciarán al primer carácter que no coincidió en este ajuste. Si el ajuste tiene éxito, se puede obtener más información usando los métodos: start(), end(), y group(). Lanza: IndexOutOfBoundsException - Si el parámetro start < 0 o si es mayor a la longitud de la secuencia de caracteres.

public boolean **lookingAt**() Intenta ajustar la secuencia de caracteres al patrón, a partir del inicio de la secuencia. Regresa true si y sólo sí un prefijo de la secuencia se ajusta al patrón. Si el ajuste tiene éxito, se puede obtener más información usando los métodos: start(), end(), y group().
public Pattern **pattern**() Regresa el patrón que es interpretado por este ajustador.
public Matcher **reset**() Restablece este matcher. Al restablecerlo se descarta toda su información explícita sobre su estado y se establece la posición de agregado a cero. El método regresa este ajustador.

Métodos de la Clase Matcher. Cont.

public Matcher **reset**(CharSequence input) Restablece este ajustador con una nueva secuencia de caracteres. Al restablecerlo se descarta toda su información explícita sobre su estado y se establece la posición de agregado a cero. El método regresa este ajustador.
public Matcher **usePattern**(Pattern newPattern) Cambia el patron usado por este ajustador para encontrar coincidencias al patrón del parámetro. El método regresa este ajustador. Este método hace que el ajustador pierda la información acerca de los grupos del último ajuste. La posición del ajustador en la secuencia de caracteres se mantiene y la última posición de agregado no se ve afectada. Lanza: IllegalArgumentException – Si el parámetro es null.
public String **replaceAll**(String replacement) Regresa una cadena en la que cada subsecuencia de la secuencia de caracteres que se ajusta al patrón es reemplazada por la cadena del parámetro. Por ejemplo, dada la expresión regular a*b, la secuencia de caracteres "aabfooaabfooabfoob" y la cadena de reemplazo "-", la invocación a este método regresa la cadena "-foo-foo-foo-".
public String **replaceFirst**(String replacement) Regresa una cadena en la que la primera subsecuencia de la secuencia de caracteres que se ajusta al patrón es reemplazada por la cadena del parámetro. Por ejemplo, dada la expresión regular can, la secuencia de caracteres "zzzcanzzzcanzzz" y la cadena de reemplazo "gato", la invocación a este método regresa la cadena "zzzgatozzzcanzzz".

public Matcher **appendReplacement**(StringBuffer sb, String replacement)

Este método hace las siguientes acciones:

1. Le agrega al StringBuffer del parámetro sb, la subsecuencia de la secuencia de caracteres desde el punto de agregado hasta antes del primer carácter de la subsecuencia que se ajusta al patrón.
2. Le agrega al StringBuffer del parámetro sb, la cadena de reemplazo dada por el parámetro replacement.
3. Establece el valor del punto de reemplazo a la siguiente posición del último carácter de la subsecuencia que se ajusta al patrón.

Lanza:

IllegalStateException - Si no se ha intentado aún ajustar la secuencia de caracteres al patrón o si no la operación de ajuste previa falló.

public StringBuffer **appendTail**(StringBuffer sb)

Este método lee caracteres de la secuencia de entrada a partir de la posición de agregado y los agrega al StringBuffer del parámetro sb.

Este método se usa normalmente junto con los métodos find() y appendReplacement() para hacer una búsqueda y reemplazo. Por ejemplo, el siguiente código le agrega al StringBuffer sb, la secuencia "un gato, dos gatos en el patio":

```
Pattern p = Pattern.compile("gato");
Matcher m = p.matcher("un perro, dos perros en el patio ");
StringBuffer sb = new StringBuffer(); while (m.find()) {
   m.appendReplacement(sb, "perro");
}
m.appendTail(sb);
```

7.3 La Clase Pattern

La clase Pattern es una representación compilada de una expresión regular.

Una cadena con una expresión regular debe compilarse a una instancia de esta clase. El patrón resultante puede crearse para crear un ajustador que pueda ajustar secuencias de caracteres a expresiones regulares.

Algunos de los atributos y métodos de la clase Pattern y que podemos usar todos los patrones del apartado anterior

Tabla Atributos de la Clase Pattern.

public static final int **CASE_INSENSITIVE** Establece que el ajuste se hará sin hacer distinción entre minúsculas y mayúsculas.
public static final int **MULTILINE** Activa el modo multilínea. En el modo multilínea los metacaracteres ^ y $ se ajustan hasta después o hasta antes de un carácter terminador de línea o el final de la secuencia de entrada, respectivamente. Por omisión los metacaracteres sólo se ajustan al principio y final de la secuencia de caracteres completa.
public static Pattern **compile**(String regex) Compila la expresión regular del parámetro a un patrón Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.

Tabla Métodos de la Clase Pattern.

public static Pattern **compile**(String regex, int flags) Compila la expresión regular del parámetro regex a un patrón con las flags dados por el parámetro flags. Dos de las banderas son: CASE_INSENSITIVE y MULTILINE. Las banderas pueden combinarse usando el operador or para bits |. Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida. IllegalArgumentException – Si una de las banderas no es válida.
public Matcher **matcher**(CharSequence input) Crea y regresa un ajustador que ajustará la secuencia de caracteres del parámetro a este patrón.

public static boolean **matches**(String regex, CharSequence input) Compila la expresión regular del parámetro regex a un patrón e intenta ajustarlo a la secuencia de caracteres del parámetro input. Regresa true si y sólo sí la secuencia completa se ajusta al patrón. La invocación a este método produce el mismo resultado que la expresión: Pattern.compile(regex).matcher(input).matches() Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.
public Pattern **pattern**() Regresa el patrón que es interpretado por este ajustador.
public Matcher **reset**() Restablece este matcher. Al restablecerlo se descarta toda su información explícita sobre su estado y se establece la posición de agregado a cero. El método regresa este ajustador.
public String **pattern**() Regresa la expresión regular de la que se compiló este patrón.

Tabla Métodos de la Clase Pattern. Cont.

Public String[] **split**(CharSequence input)

Separa la secuencia de caracteres del parámetro en subcadenas tomando las coincidencias con este patrón como separadores. El método regresa un arreglo con las subcadenas obtenidas. Las subcadenas en el arreglo aparecen en el orden en que ocurrieron en la secuencia de caracteres. Si este patrón no se ajusta a ninguna subsecuencia de la secuencia de caracteres, entonces el arreglo sólo tendrá un elemento, la secuencia de caracteres completa como una cadena. Las cadenas vacías al final del arreglo serán descartadas.

Por ejemplo, la secuencia de caracteres "boo:and:foo" produce los siguiente resultados con esas expresiones regulares:

Expresión Regular	Contenido del Arreglo
:	{"boo", "and", "foo"}
o	{"b", "", ":and:f"}

public String[] **split**(CharSequence input, int limit)

Separa la secuencia de caracteres del parámetro en subcadenas tomando las coincidencias con este patrón como separadores. El método regresa un arreglo con las subcadenas obtenidas. Las subcadenas en el arreglo aparecen en el orden en que ocurrieron en la secuencia de caracteres. Si este patrón no se ajusta a ninguna subsecuencia de la secuencia de caracteres, entonces el arreglo sólo tendrá un elemento, la secuencia de caracteres completa como una cadena.

El parámetro limit controla el número de veces que se aplica el patrón y por lo tanto afecta al tamaño del arreglo. Si el valor de limit < 0, entonces el patrón se aplicará cuando mucho limit -1 veces, la longitud del arreglo no será mayor a limit, y el último elemento del arreglo contendrá la subsecuencia con todos los caracteres después de la última coincidencia. Si limit es negativo, el patrón se aplicará tantas veces como sea posible y el arreglo tendrá esa longitud. Si limit es cero, el patrón se aplicará tantas veces como sea posible, el arreglo podrá tener cualquier longitud y las cadenas vacías al final del arreglo serán descartadas.

Por ejemplo, la secuencia de caracteres "boo:and:foo" produce los siguiente resultados con esas expresiones regulares:

Expresión Regular	limit	Contenido del Arreglo
:	2	{"boo", "and:foo"}
:	5	{"boo", "and", "foo"}
:	-2	{"boo", "and", "foo"}
o	5	{"b", "", ":and:f", "", ""}
o	-2	{"b", "", ":and:f", "", ""}
o	0	{"b", "", ":and:f"}

7.4 La Excepción PatternSyntaxException

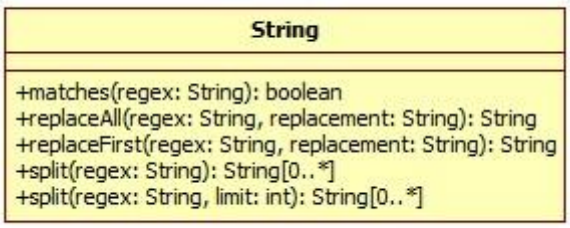
Esta excepción no verificada se utiliza para indicar errores en los patrones de expresiones regulares. Esta excepción hereda de IllegalArgumentException que a su vez hereda de RuntimeException. Los métodos adicionales de esta clase

Excepción PatternSyntaxException.

public **PatternSyntaxException**(String desc, String regex, int index) Construye una nueva instancia de esta clase con la descripción del error, la expresión regular con el error y el índice aproximado del error o -1 si no se conoce la posición.
public int **getIndex**() Regresa el índice aproximado del error o -1 si no se conoce la posición.
public String **getDescription**() Regresa la descripción del error.
public String **getPattern**() Regresa la expresión regular con el error.
public String **getMessage**() Regresa una cadena multilínea con la descripción del error de sintaxis y su índice. La expresión regular inválida y una indicación visual del índice del error en la expresión. Este método sobrescribe el método getMessage() de la clase Throwable.

7.5 La clase `String` y Las Expresiones Regulares

La clase String de la API de Java tiene un conjunto de métodos que nos permite realizar algunas de las operaciones con expresiones regulares. Esos métodos se muestran en el diagrama de clases de la figura 11.2 y en la tabla 11.13.



Métodos de la Clase String para Expresiones Regulares.

Métodos de la Clase String para Expresiones Regulares.

public boolean **matches**(String regex) Regresa true si y sólo sí esta cadena se ajusta a la expresión regular de su parámetro. La invocación a este método produce el mismo resultado que la expresión: Pattern.matches(regex, str) Donde str es la cadena de esta clase. Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.
public String **replaceAll**(String regex, String replacement) Regresa una cadena en la que cada subcadena de esta cadena que se ajusta a la expresión regular del parámetro regex es reemplazada por la cadena del parámetro replacement. La invocación a este método produce el mismo resultado que la expresión: Pattern.compile(*regex*).matcher(*str*).replaceAll(replacement) Donde str es la cadena de esta clase. Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.
public String **replaceFirst**(String regex, String replacement) Regresa una cadena en la que la primera subcadena de esta cadena que se ajusta a la expresión regular del parámetro regex es reemplazada por la cadena del parámetro replacement. La invocación a este método produce el mismo resultado que la expresión: Pattern.compile(*regex*).matcher(*str*).replaceFirst(replacement) Donde str es la cadena de esta clase. Lanza: PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.

Métodos de la Clase String para Expresiones Regulares.

public String[] **split**(String regex)

Separa la secuencia de caracteres del parámetro en subcadenas tomando las coincidencias con este patrón como separadores. El método regresa un arreglo con las subcadenas obtenidas. Las subcadenas en el arreglo aparecen en el orden en que ocurrieron en la secuencia de caracteres. Si este patrón no se ajusta a ninguna subsecuencia de la secuencia de caracteres, entonces el arreglo sólo tendrá un elemento, la secuencia de caracteres completa como una cadena. Las cadenas vacías al final del arreglo serán descartadas.

La invocación a este método produce el mismo resultado que la expresión:

```
Pattern.compile(regex).split(str)
```

Donde str es la cadena de esta clase.

Lanza:

PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.

public String[] **split**(String regex, int limit)

Separa la secuencia de caracteres del parámetro en subcadenas tomando las coincidencias con este patrón como separadores. El método regresa un arreglo con las subcadenas obtenidas. Las subcadenas en el arreglo aparecen en el orden en que ocurrieron en la secuencia de caracteres. Si este patrón no se ajusta a ninguna subsecuencia de la secuencia de caracteres, entonces el arreglo sólo tendrá un elemento, la secuencia de caracteres completa como una cadena.

El parámetro limit controla el número de veces que se aplica el patrón y por lo tanto afecta al tamaño del arreglo. Si el valor de limit < 0, entonces el patrón se aplicará cuando mucho limit -1 veces, la longitud del arreglo no será mayor a limit, y el último elemento del arreglo contendrá la subsecuencia con todos los caracteres después de la última coincidencia. Si limit es negativo, el patrón se aplicará tantas veces como sea posible y el arreglo tendrá esa longitud. Si limit es cero, el patrón se aplicará tantas veces como sea posible, el arreglo podrá tener cualquier longitud y las cadenas vacías al final del arreglo serán descartadas.

```
Pattern.compile(regex).split(str, limit)
```

Donde str es la cadena de esta clase.

Lanza:

PatternSyntaxException - Si la sintaxis de la expresión regular es inválida.

7.6 Practica guiada expresiones regulares

Evaluamos diferentes expresiones regulares de los ejercicios anteriores. Fundamentalmente vamos a probar búsquedas validaciones y alguna sustitucion

Vamos a explicar las partes marcadas en amarillo y en azul que son las claves en el código.

Declaramos una cadena

```
String cadena = "dasfabcsdfds";
```

En la variable pat de tipo pattern guardamos la expresión regular compilada, es la representación de la expresión en Java. El método estático compile de la clase Pattern nos devuelve la expresion regular compilada en un objeto de tipo Pattern.

```
Pattern pat = Pattern.compile("abc");
```

Para hacer las comprobaciones usamos la clase Matcher, fundamentalmente dos métodos:

1. El método matches para ver si la cadena entera cumple el patrón.
2. El método find para buscar el patrón en la cadena.

```
Matcher mat = pat.matcher(cadena);

if (mat.matches())
```

Además con los métodos start y end podemos saber donde empieza y termina el patrón de la cadena

```
if (mat2.find()) {

                    System.out.print("Start index: " + mat2.start());
                   System.out.print(" End index: " + mat2.end() + " ");
```

Notas Cornell:

En el **siguiente ejemplo validamos un dni con matches. Marcar con los marcadores que métodos estamos usando y que hace este código**. Explicarlo con vuestras propias palabras.

```
Pattern patdni = Pattern.compile("^[0-9]{8}");
            Matcher matdni = pat.matcher(dni);
            if (matdni.matches()) {
```

¿En el siguiente ejemplo que buscamos? Explicar la expresión regular y realizar los mismos pasos que con el ejemplo de DNI

```
Pattern patAs = Pattern.compile("\\w*\\B(as)\\B\\w*");
            Matcher matAs = patAs.matcher(cadenasAS);
            if (matAs.matches()) {
```

PatterYMatcherMatchAndFind.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternYMatcherMatchAndFind {

      public static void main(String[] args) {
            // TODO Auto-generated method stub

            // Ejemplos de Expresiones Regulares en Java:1. Comprobar si el
String cadena
            // contiene exactamente el patrón (matches) “abc”Pattern

            String cadena = "dasfabcsdfds";

            Pattern pat = Pattern.compile("abc");
            Matcher mat = pat.matcher(cadena);
            if (mat.matches()) {


                  System.out.println("SI");
            } else {
                  System.out.println("NO");
            }
            // Comprobar si el String cadena contiene “abc”
            //
            Pattern.compile(".*abc.*");
            Matcher mat2 = pat.matcher(cadena);
            if (mat2.find()) {

                  System.out.print("Start index: " + mat2.start());
                 System.out.print(" End index: " + mat2.end() + " ");
                 System.out.println(mat2.group());
                  System.out.println("SI");
```

```
		} else {
			System.out.println("NO");
		} // También lo podemos escribir usando el método find:

		// Validar un DNI
		String dni = "44567894";

		Pattern patdni = Pattern.compile("^[0-9]{8}");
		Matcher matdni = pat.matcher(dni);
		if (matdni.matches()) {
			System.out.println("EL DNI es valido");

		} else {
			System.out.println("EL DNI es valido");
		}


		//  Comprobar si el String cadena contiene “as”
		String cadenasAS = "adadasdfd";

		Pattern patAs = Pattern.compile("\\w*\\B(as)\\B\\w*");
		Matcher matAs = patAs.matcher(cadenasAS);
		if (matAs.matches()) {
			System.out.println("Contiene as");

		} else {
			System.out.println("No contiene as");
		}



	}

}
```

Las expresiones regulares son muy útiles también para reemplazar o separar cadenas. Vamos a realizar un ejemplo de reemplazar. E

```
package expresionesregulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternYMatcherReplacementSplit {

	public static void main(String[] args) {
		// TODO Auto-generated method stub


		// reemplazar “as” por "bt"

				String cadenasASREP = "adadasdasdfdfasfd";

				Pattern patAsRep = Pattern.compile("\\B(as)\\B");
```

```
                        Matcher matAsRep = patAsRep.matcher(cadenasASREP);

                        String resultado = matAsRep.replaceAll("bt");

                        System.out.println("Cadena original " +
cadenasASREP);
                        System.out.println("Resultado de reemplazar as: " +
resultado);


                        // reemplazar “as” por "bt" con la clase String

                        String cadenasASREPString = "adadasdasdfdfasfd";
                        String resultadoString =
cadenasASREPString.replaceAll("\\B(as)\\B", "bt");


                        System.out.println("Cadena original " +
cadenasASREPString);
                        System.out.println("Resultado de reemplazar as: " +
resultadoString);



    }

}
```

7.7 Tarea de refuerzo

Se pide realizar un programa que:

1. Pida el **dni de la persona**
2. Pida el **número de matrícula** de su vehículo habitual
3. Pida su correo electrónico
4. Valide el **DNI y la matricula** e indique si son correctos
5. Valide el **correo electrónico** teniendo en cuenta que:
    a. Contenga @
    b. Acabe en el formato dominio .dominio, por ejemplo .com

8 Bibliografía y referencias web

Referencias web

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021

Entornos de Desarrollo, Maria Jesus Ramos Martín, Garceta 2ª Edición, 2020

PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS, Enrique Quero Catalinas y José López Herranz, Paraninfo, 1997.