Material descargable

egg

Teoría **JAVA II**

¡Hola nuevamente! 👋

En esta oportunidad, continuaremos nuestro viaje de aprendizaje en el fascinante mundo de la programación backend con Java. En esta ocasión, nos enfocaremos en las *estructuras de control* y las *clases fundamentales* que te permitirán ampliar tus habilidades como programador/a Java.

Exploraremos las **estructuras de control**, como el *if*, *if-else*, *if-else-if* y *switch*, para tomar decisiones y controlar el flujo de ejecución de tu programa. También profundizaremos en el manejo de excepciones con *try-catch*, permitiéndote capturar y manejar errores de manera controlada.

Además, abordaremos el **ámbito de las variables**, tomando decisiones inteligentes al definirlas y acceder a ellas en diferentes partes de tu programa.

Por último, nos sumergiremos en las **clases String y Math**, que ofrecen funcionalidades poderosas para trabajar con cadenas de texto y realizar operaciones matemáticas avanzadas.

¡Comencemos! 🚀

Estructuras de control

Las **estructuras de control** en Java son mecanismos que nos permiten controlar el flujo de ejecución de un programa.

Estas estructuras nos brindan la capacidad de tomar decisiones y ejecutar diferentes bloques de código según ciertas condiciones. También nos permiten manejar excepciones y controlar situaciones inesperadas durante la ejecución de nuestro programa.

Hasta ahora, nuestro código se ha ejecutado secuencialmente, es decir, línea por línea. Sin embargo, las estructuras de control nos brindan la capacidad de

ejecutar bloques de código específicos en función de condiciones que nosotros establezcamos. **Esto nos permite tener un mayor control y flexibilidad en la ejecución de nuestro programa**.

If

El **“if”** es una *estructura de control condicional* que nos permite ejecutar un bloque de código si se cumple una condición determinada.

Veamos un ejemplo:

```
public static void main(String[] args) {
        //Sentencias que se ejecutan antes del “if”
        if (condición) {
            /*
            Sentencias que se ejecutan si la condición es verdadera.
            Si la condición es falsa, el código dentro del bloque “if” no se
ejecuta.
            */
        }
        //Sentencias que se ejecutan después del “if”
    }
```

La *sintaxis* comienza con la palabra clave *"if"* seguida de unos paréntesis que contienen la *"condición"*.

La condición es una expresión booleana que puede devolver true o false.

💡 ***Recuerda que podemos construir expresiones booleanas utilizando operadores de comparación y operadores lógicos.***

Después de la condición, se utilizan llaves para delimitar el bloque de código que se ejecutará si la condición es verdadera.

If-else

El **"if-else"** es una estructura de control condicional que nos permite ejecutar un bloque de código si se cumple una condición determinada, y otro bloque de código si la condición no se cumple.

Veamos un ejemplo:

```
public static void main(String[] args) {
        //Sentencias que se ejecutan antes del “if else”.
        if (condición) {
            /*
            Sentencias que se ejecutan si la condición es verdadera.
            Si la condición es falsa, el código dentro del bloque “if” no se
ejecuta.
            */
        } else {
            /*
            Sentencias que se ejecutan si la condición es falsa.
            Si la condición es verdadera, el código dentro del bloque “else”
no se ejecuta.
            */
        }
        //Sentencias que se ejecutan después del “if else”.
    }
```

La *sintaxis* incluye la palabra clave *"if"* seguida de una condición entre paréntesis:

- Si la condición es *verdadera*, se ejecuta el bloque de código dentro del *"if"*.
- Si la condición es *falsa*, se ejecuta el bloque de código dentro del *"else"*.

Después de cada bloque de código, se continúa con las sentencias que se ejecutan después del *"if-else"*.

If-else-if

El **"if-else-if"** es una estructura de control condicional que nos permite ejecutar un bloque de código si se cumple una condición determinada y otro bloque de código sólo si se cumple otra condición.

```java
public static void main(String[] args) {
        //Sentencias que se ejecutan antes del “if else if”.
        if (condición1) {
            /*
            Sentencias que se ejecutan si la condición1 es verdadera.
            Si la condición es falsa, el código dentro del bloque “if” no se
ejecuta.
            */
        } else if (condición2) {
            /*
            Sentencias que se ejecutan si la condición1 es falsa y la
condicion2 es verdadera.
            Si la condición1 es verdadera o la condición2 es falsa, el
código dentro del bloque
            “else if” no se ejecuta.
            */
        }
        //Sentencias que se ejecutan después del “if else if”.
    }
```

Después de la llave de cierre del bloque *"else if (condición2)*", podemos agregar otro bloque *"else"* o *"else if (condicion)"* tantas veces como queramos.

Recuerda que en cada bloque condicional, *solo se ejecutará el código correspondiente si la condición asociada es verdadera*. Si ninguna de las condiciones es verdadera, se omitirán todos los bloques condicionales y se ejecutarán las sentencias después del *"if-else-if"*.

Switch

El **"switch"** es una *estructura de control que nos permite seleccionar uno de varios bloques de código para ejecutar*, dependiendo del valor de una expresión o variable.

La expresión dentro del *switch* se evalúa y se compara con los casos definidos dentro de él. Cada caso representa una opción diferente, y se ejecutará el bloque de código correspondiente si el valor de la expresión coincide con el caso.

Para comprenderlo mejor, veamos el siguiente ejemplo:

```
public static void main(String[] args) {
        //Sentencias que se ejecutan antes del "switch".
        switch (opcion) {
            case 1:
                //Sentencias que se ejecutan si "opción" tiene el "valor 1".
                System.out.println("Seleccionaste la opción 1");
                break; //Palabra clave "break" para salir del "switch".
            case 2:
                //Sentencias que se ejecutan si "opción" tiene el "valor 2".
                System.out.println("Seleccionaste la opción 2");
                break;
            case 3:
                //Sentencias que se ejecutan si "opción" tiene el "valor 3".
                System.out.println("Seleccionaste la opción 3");
                break;
            default:
                //Sentencias que se ejecutan si "opción" no coincide con el
                valor de ningún "case".
                System.out.println("Opción inválida");
                break;
        }
        //Sentencias que se ejecutan después del "switch".
    }
```

Ten en cuenta que *"opcion"* puede ser una variable de tipo texto también. La palabra reservada *"break"* se utiliza para salir del switch después de ejecutar el bloque correspondiente a un caso. Sin el "break", el código continuaría ejecutando los casos siguientes, incluso si no coinciden con el valor de la expresión.

Aquí te compartimos otro ejemplo donde podemos agrupar varios casos para ejecutar la misma línea de código:

```
public static void main(String[] args) {
        switch (opcion) {
            case "A":
            case "B":
            case "C":
                System.out.println("Las opciones A, B y C están
                deshabilitadas");
                break;
            case "D":
                System.out.println("Seleccionaste la opción D");
                break;
            case "E":
                System.out.println("Seleccionaste la opción E");
                break;
            default:
                System.out.println("Opción inválida");
                break;
        }
    }
```

Switch expression

"Switch expression" es una forma nueva y mejorada de utilizar el switch que se introdujo en Java 12. Esta nueva característica *proporciona una sintaxis más flexible y mejorada que permite evaluar diferentes tipos de datos, incluyendo cadenas de texto (strings), y utilizar expresiones más complejas*.

Veamos un ejemplo:

```
public static void main(String[] args) {
    //Sentencias que se ejecutan antes del "switch".
     switch (opcion) {
        case 1  -> System.out.println("Seleccionaste la opción 1");
        case 2  -> System.out.println("Seleccionaste la opción 2");
        case 3  -> System.out.println("Seleccionaste la opción 3");
        default -> System.out.println("Opción inválida");
     }
    //Sentencias que se ejecutan después del "switch".
}
```

En este nuevo enfoque, la sintaxis es más concisa y *no se requiere el uso de la palabra reservada "break"*. Cada caso se representa con una *flecha (->)* seguida

del *bloque de código* que se ejecutará si el valor de la expresión coincide con el caso (“case”).

Es importante tener en cuenta que en el Switch Expression, si deseas incluir varias líneas de código para un caso determinado, debes utilizar *llaves ({})* para agruparlas, como se muestra en el siguiente ejemplo:

```
public static void main(String[] args) {
        //Sentencias que se ejecutan antes del "switch".
        switch (opcion) {
            case 1  -> {
                System.out.print("Seleccionaste la opción:");
                System.out.print(" 1");
           }
            case 2  -> System.out.println("Seleccionaste la opción 2");
            case 3  -> System.out.println("Seleccionaste la opción 3");
            default -> System.out.println("Opción inválida");
        }
        //Sentencias que se ejecutan después del "switch".
    }
```

En este caso, el *“case 1”* tiene varias líneas de código y se agrupan utilizando *llaves ({})* para delimitar el bloque que se ejecutará.

Switch como expresión

Como su nombre lo indica, **el switch como expresión es un nuevo enfoque que permite que el switch funcione como una expresión**, lo cual significa que puede calcular directamente un valor. *Antes de esta actualización, el switch solo se podía utilizar como una declaración.*

Para comprender esta diferencia, analicemos cómo se modificaría el valor de una variable utilizando el switch statement tradicional y cómo se haría con el switch como expresión.

Imaginemos que deseamos obtener la cantidad de días de un mes a partir de su nombre. Con el switch statement tradicional, tendríamos que declarar la variable antes del switch y asignarle un valor dentro de cada case:

```java
public static void main(String[] args) {
        String mes = "February";
        int numeroDeDias;
        switch (mes) {
            case "February":
                numeroDeDias = 28;
                break;
            case "April":
            case "June":
            case "September":
            case "November":
                numeroDeDias = 30;
                break;
            case "January":
            case "March":
            case "May":
            case "July":
            case "August":
            case "October":
            case "December":
                numeroDeDias = 31;
                break;
            default:
                numeroDeDias = 0;
        }
        System.out.println(mes+" tiene "+numeroDeDias+" días.");
}
```

En cambio, con el switch como expresión, podemos asignar directamente el resultado del switch a la variable, sin necesidad de declararla previamente:

```java
public static void main(String[] args) {
        String mes = "February";
        int numeroDeDias = switch (month) {
            case "February" -> 28;
            case "April", "June", "September", "November" -> 30;
            case "January", "March", "May", "July", "August", "October", "December" ->
31;
            default -> 0;
        };
        System.out.println(mes+" tiene "+numeroDeDias+" días.");
}
```

En este último caso, utilizamos el switch como expresión para asignar directamente el resultado del cálculo de la cantidad de días a la variable "numeroDeDias". Esto nos permite simplificar el código y hacerlo más conciso.

Yield

Al utilizar bloques de código en una *expresión switch* para manejar múltiples líneas de código, **se emplea la palabra clave "yield" para indicar el valor de retorno del case**.

A continuación, te mostramos un ejemplo de código que utiliza esta estructura:

```
public static void main(String[] args) {
        String position = "director";
        boolean alcanzoObjetivos = true;
        double bonus = switch (position) {
            case "temporal" -> 50;
            case "empleado" -> 1000;
            case "director" -> {
                double bonusBase = 2000;
                double bonusPorRendimiento = alcanzoObjetivos ? 500 : 0;
                double bonusTotal = bonusBase + bonusPorRendimiento;
                yield bonusTotal;
            }
            default -> 0;
        };
        System.out.println("El bonus del "+position+" es $"+bonus);
}
```

En el programa, se define una variable *"position"* con el valor *"director"* y una variable booleana *"alcanzoObjetivos"* con el valor *verdadero*. Luego, se realiza una *expresión switch* basada en la variable *"position"*. En el *case "director"*, se realiza un cálculo de bonificación que incluye un bono base y un bono adicional basado en si se alcanzaron los objetivos. El resultado se asigna a la variable *"bonusTotal"* y se utiliza la *palabra clave "yield"* para devolver dicho valor. En caso de no coincidir con ningún *case*, se asigna el valor 0. Finalmente, se imprime en pantalla el resultado del cálculo del bonus correspondiente a la posición *"director"*.

Exhaustividad

La **exhaustividad** en una *expresión switch* es un concepto fundamental que implica cubrir todos los casos (“cases”) posibles. Si conoces de antemano todos los valores que puede tomar tu variable, es necesario tener un case correspondiente para cada uno de ellos. En caso de no conocer todos los posibles valores, es recomendable incluir una *cláusula default* para manejar cualquier valor no previsto.

💡 *Es importante destacar que este principio es especialmente relevante al trabajar con tipos de datos como **int** o **string**, los cuales pueden tener un rango de valores amplio o indefinido.*

Ahondaremos más sobre este tema cuando veamos estructuras de datos con un rango de valores más acotado, como los Enums.

Coincidencia de patrones

En las versiones más recientes de Java, se ha introducido una característica muy útil conocida como **"Pattern matching for switch" (coincidencia de patrones para switch)**.

Esta funcionalidad **proporciona una forma más eficiente y legible de manejar diferentes tipos de objetos en nuestros programas**, lo que puede mejorar significativamente nuestro código. Sin embargo, es importante tener en cuenta que para comprender y utilizar eficazmente esta característica, es necesario tener un buen entendimiento de otros conceptos de programación orientada a objetos en Java.

Aunque no profundizaremos en el *“pattern matching for switch”* en este momento, es relevante destacar su existencia y su estudio detallado más adelante en el curso, una vez que hayamos abordado los conceptos necesarios.

Bloque try-catch

El **“bloque try-catch”** en Java *se utiliza para manejar excepciones*.

Se coloca el código propenso a errores o que puede generar excepciones dentro del *"bloque try"*. Si ocurre una excepción dentro del *"bloque try"*, se captura en uno o varios *"bloques catch"*, donde se especifica el tipo de excepción que se desea manejar y el código correspondiente para tratar la excepción.

Para entender cómo funciona, observemos los siguientes dos ejemplos y lo que se muestra por consola:

```java
public static void main(String[] args) {
        Scanner pepe = new
Scanner(System.in);
        System.out.print("Ingrese un número
divisor de 10: ");
        int numero = pepe.nextInt();
        try {
            double resultado = 10/numero;
            // Posible división por cero
            System.out.println("El resultado
es: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error: No es
posible dividir por 0.");
        }
        System.out.println("¡Gracias! ");
}
```

```
CONSOLA:
Ingrese un número divisor de 10: 0
Error: No es posible dividir por 0.
¡Gracias!
```

```java
public static void main(String[] args) {
       Scanner pepe = new
Scanner(System.in);
        System.out.print("Ingrese un número
divisor de 10: ");
        int numero = pepe.nextInt();
        double resultado = 10/numero;
        // Posible división por cero
        System.out.println("El resultado es:
" + resultado);
        System.out.println("Gracias! ");
}
```

```
CONSOLA:
Ingrese un número divisor de 10: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at
EjemplosTeoria.main(EjemplosTeoria.java:
8)
```

En los dos ejemplos anteriores, se puede observar que sin el *"bloque try-catch"*, nuestro programa se detiene en la línea que genera el error *"double resultado = 10/numero;"*, impidiendo que la ejecución continúe.

En contraste, al utilizar el *"bloque try-catch"*, el programa salta las líneas de código dentro del *"bloque try"* cuando se encuentra con una línea que produce un error. En lugar de detenerse, continúa ejecutando el programa a partir del *"bloque catch"*, permitiendo que las líneas de código restantes se ejecuten después del bloque.

Manejo de excepciones

En Java, **existen diversos tipos de excepciones y es posible declarar diferentes *bloques catch* para capturarlas** y realizar acciones específicas para cada una de ellas.

En el siguiente ejemplo de código, ilustramos este concepto:

```
public static void main(String[] args) {
        try {
            Scanner pepe = new Scanner(System.in);
            System.out.print("Ingrese un divisor: ");
            int numero = pepe.nextInt();// Posible entrada inválida
            String palabra = "hola";
            double resultado = 10 / numero ;// Posible división por cero
            System.out.println("El resultado es: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        } catch (InputMismatchException e) {
            System.out.println("Error: Se detectó un valor inválido ingresado
por teclado.");
        } catch (Exception e) {
            System.out.println("Error: Ups!");
        }
 }
```

El *"InputMismatchException"* ocurre cuando se produce un error al intentar convertir la entrada del usuario a un tipo de dato de Java, por ejemplo, si el usuario ingresa letras cuando se espera un número.

En el ejemplo anterior, se utiliza *"Exception"* como un bloque catch adicional que atrapa cualquier excepción no especificada en los bloques catch anteriores.

Es importante colocar el *bloque catch de Exception* al final, ya que de lo contrario, este bloque interceptaría cualquier excepción antes de que los bloques catch específicos pudieran ser utilizados.

```
public static void main(String[] args) {
        try {
            // Código que puede lanzar una excepción
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Error: Ups!: " + e.getMessage());
        }
 }
```

Dentro de los bloques catch, podemos observar la *variable "e"*, que es del tipo de la excepción capturada. *Esta variable nos proporciona acceso a métodos útiles que podemos utilizar.*

A continuación, veremos dos de ellos:

- **printStackTrace():** Este método imprime por consola la *"pila de llamadas de la excepción"*, mostrando la información sobre el lugar exacto donde ocurrió el problema. Esto resulta especialmente útil en aplicaciones más complejas, ya que nos permite identificar y solucionar errores de manera más efectiva. Por ejemplo:

```
java.util.InputMismatchException
      at java.base/java.util.Scanner.throwFor(Scanner.java:947)
      at java.base/java.util.Scanner.next(Scanner.java:1602)
      at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
      at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
      at EjemploTeoria.main(EjemploTeoria.java:9)
```

  En la imagen podemos observar la salida de un *"printStackTrace()"* dentro de un *bloque catch* que captura una *Excepción "InputMismatchException"*. Destacado en azul, se puede ver que se menciona la clase *"EjemploTeoria"* donde ocurre la excepción, así como el método main y la línea de código número 9 del archivo.

- **getMessage():** Este método devuelve únicamente el mensaje de descripción del error, sin incluir la pila de llamadas. Puede ser utilizado para complementar mensajes personalizados de error. En futuros temas, exploraremos otros usos que se le pueden dar a este método.

💡 *En resumen, comprender y utilizar adecuadamente el manejo de excepciones en Java nos permite capturar y tratar diferentes tipos de errores en nuestros programas, mejorando su robustez y facilitando la identificación y corrección de problemas.*

Ámbito de las variables

En Java, **las variables que se declaran dentro de un bloque tienen un ámbito limitado a ese bloque en particular**. Esto implica que solo son visibles y accesibles dentro de dicho bloque y no se pueden acceder desde fuera de él. Una vez que se sale del bloque, esas variables dejan de existir y no se pueden utilizar.

💡 *Recuerda que un bloque de código es aquel que se encuentra entre llaves "{}".*

Hasta ahora, esto no nos había afectado porque trabajábamos exclusivamente dentro del bloque de código del método main. Sin embargo, *con la incorporación de estructuras de control, surgen nuevos bloques de código, por lo tanto, debemos tener cuidado al declarar nuestras variables* si posteriormente deseamos utilizarlas fuera de los bloques que creamos.

Observemos los siguientes ejemplos:

`public static void main(String[] args) {`
`  int dato1 = pepe.nextInt();`
`  int dato2 = pepe.nextInt();`
`  if (dato1 != 0 && dato2 != 0) {`
`  int resultado = dato1 + dato2;`
`  }`
`  System.out.println(resultado);`
`}`
	`public static void main(String[] args) {`
	`  int dato1 = pepe.nextInt();`
	`  int dato2 = pepe.nextInt();`
	`  int resultado;`
	`  if (dato1 != 0 && dato2 != 0) {`
	`  resultado = dato1 + dato2;`
	`  }`
	`  System.out.println(resultado);`
	`}`

La línea System.out.println(resultado); marca un error porque la variable "resultado" no existe dentro del ámbito del método main. Se declara dentro del bloque if y no se puede acceder a ella fuera de ese bloque.	El programa se compila correctamente porque la variable "resultado" se declara dentro del ámbito del método main. De esta manera, se puede acceder a ella fuera del bloque if.

Clase String

La **clase String** es una clase fundamental que **se utiliza para representar y manipular cadenas de caracteres**. Es una de las clases más utilizadas en Java y proporciona una amplia gama de métodos para realizar operaciones comunes en cadenas, como la concatenación, búsqueda, reemplazo, extracción y manipulación de caracteres.

Te compartimos los siguientes ejemplos de algunos *métodos comunes* de la *clase String*:

- **length():** Devuelve la longitud de la cadena, es decir, el número de caracteres que contiene.

```
String texto = "Hola Mundo";
int longitud = texto.length(); // Devuelve 10
```

- **charAt(int index):** Devuelve el carácter en la posición especificada por el índice.

```
String texto = "Hola Mundo";
char primerCaracter = texto.charAt(0); // Devuelve 'H'
```

- **substring(int beginIndex, int endIndex):** Devuelve una subcadena de la cadena original, desde el índice de inicio hasta el índice de fin (excluido).

```
String texto = "Hola Mundo";
String subcadena = texto.substring(5, 10); // Devuelve "Mundo"
```

- **equals(String str):** Compara dos cadenas y devuelve true si son iguales, es decir, si contienen los mismos caracteres en el mismo orden.

```
String texto1 = "Hola";
String texto2 = "Hola";
boolean sonIguales = texto1.equals(texto2); // Devuelve true
```

- **indexOf(String str):** Devuelve el índice de la primera aparición de la subcadena especificada dentro de la cadena original.

```
String texto = "Hola Mundo";
int indice = texto.indexOf("Mundo"); // Devuelve 5
```

💡 *Para obtener más información sobre los distintos métodos, puedes consultar el* 👉***Anexo A*** *al final del documento. En dicho anexo, se explican brevemente algunos de los métodos disponibles en la clase String.*

Clase Math

La **clase Math** en Java es una herramienta fundamental que **proporciona métodos y constantes para realizar operaciones matemáticas comunes**.

Una de las ventajas de esta clase es que no es necesario crear un objeto, ya que todos sus métodos son estáticos y se invocan directamente desde la clase. Además, *la clase Math pertenece al paquete java.lang, por lo que no requiere importarse* (basta con escribir *"Math."* seguido del método que deseemos utilizar para acceder a sus funcionalidades).

Además de los métodos, la clase Math incluye dos constantes matemáticas muy importantes: **PI** y **E**. Ambas constantes son de tipo double y pueden ser utilizadas en cálculos matemáticos con precisión:

```
Math.PI; //3.141592653589793
Math.E;  //2.718281828459045
```

Algunos de los métodos más comunes de la *clase Math* incluyen:

- **Math.abs():** Devuelve el valor absoluto de un número.

```
int num = -10;
int absNum = Math.abs(num); // absNum será igual a 10
```

- **Math.sqrt():** Calcula la raíz cuadrada de un número.

```
double num = 16;
double sqrtNum = Math.sqrt(num); // sqrtNum será igual a 4.0
```

- **Math.pow():** Calcula la potencia de un número.

```
double base = 2;
double exponente = 3;
double resultado = Math.pow(base, exponente); // resultado será igual a 8.0
```

- **Math.random():** Genera un número aleatorio entre 0.0 (inclusive) y 1.0 (exclusivo).

```
double randomNum = Math.random(); // Genera un número aleatorio entre 0.0 y 1.0
```

- **Math.round():** Realiza el redondeo de un número al entero más cercano.

```
double num = 3.6;
long roundedNum = Math.round(num); // roundedNum será igual a 4
```

- **Math.floor():** Redondea hacia abajo un número decimal al entero más cercano.

```
double num = 4.9;
double flooredNum = Math.floor(num); // flooredNum será igual a 4.0
```

- **Math.ceil():** Redondea hacia arriba un número decimal al entero más cercano.

```
double num = 2.2;
double ceiledNum = Math.ceil(num); // ceiledNum será igual a 3.0
```

💡 *Para obtener más información sobre los distintos métodos, puedes consultar el 👉**Anexo B** al final del documento. En dicho anexo, se explican brevemente algunos de los métodos disponibles en la clase Math.*

Anexo A

En este anexo, puedes ver algunos de los distintos métodos que tiene la **clase String** explicados brevemente:

- **charAt(int index):** Devuelve el carácter en la posición específica (índex).
- **chars():** Devuelve una secuencia de valores int que representan los caracteres de la cadena en orden.
- **codePointAt(int index):** Devuelve el punto de código Unicode en la posición especificada.
- **codePointBefore(int index):** Devuelve el punto de código Unicode inmediatamente antes de la posición especificada.
- **codePointCount(int beginIndex, int endIndex):** Devuelve la cantidad de puntos de código Unicode en un subintervalo de esta cadena.
- **codePoints():** Devuelve una secuencia de puntos de código Unicode en la cadena.
- **compareTo(String anotherString):** Compara alfabeticamente esta cadena con otra cadena.
- **compareToIgnoreCase(String str):** Compara lexicográficamente dos cadenas, ignorando las diferencias de mayúsculas y minúsculas.
- **concat(String str):** Concatena la cadena especificada al final de esta cadena.
- **endsWith(String suffix):** Comprueba si esta cadena termina con el sufijo especificado.
- **equals(Object anObject):** Compara esta cadena con el objeto especificado.
- **equalsIgnoreCase(String anotherString):** Compara esta cadena con otra cadena, ignorando las diferencias de mayúsculas y minúsculas.
- **formatted(Object... args):** Formatea la cadena utilizando los argumentos proporcionados.
- **getBytes(), getBytes(Charset charset), getBytes(String charsetName):** Convierte la cadena en una secuencia de bytes utilizando el conjunto de caracteres predeterminado o ingresado por parámetro.
- **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin):** Copia los caracteres de esta cadena en el array de destino.

- **isBlank():** Devuelve true si la cadena está vacía o contiene solo espacios en blanco.
- **isEmpty():** Devuelve true si la longitud de la cadena es 0.
- **lastIndexOf(int ch), lastIndexOf(String str), lastIndexOf(int ch, int fromIndex), lastIndexOf(String str, int fromIndex):** Devuelve el índice de la última aparición del carácter o cadena especificada, comenzando la búsqueda hacia atrás desde el índice especificado.
- **length():** Devuelve la longitud de esta cadena.
- **lines():** Devuelve una secuencia de líneas de la cadena, separadas por saltos de línea.
- **matches(String regex):** Indica si esta cadena coincide con la expresión regular dada.
- **offsetByCodePoints(int index, int codePointOffset):** Devuelve el índice en esta cadena que es desplazado desde el índice dado por el número de puntos de código.
- **regionMatches(int toffset, String other, int ooffset, int len), regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len):** Prueba si dos regiones de cadenas son iguales.
- **repeat(int count):** Devuelve una cadena cuyo valor es la concatenación de esta cadena repetida count veces.
- **replace(char oldChar, char newChar), replace(CharSequence target, CharSequence replacement):** Devuelve una nueva cadena que resulta de reemplazar todas las apariciones del carácter o la secuencia de caracteres de destino en esta cadena con el carácter o la secuencia de caracteres de reemplazo.
- **replaceAll(String regex, String replacement):** Reemplaza cada subcadena de esta cadena que coincide con la expresión regular dada con la cadena de reemplazo.
- **replaceFirst(String regex, String replacement):** Reemplaza la primera subcadena de esta cadena que coincide con la expresión regular dada con la cadena de reemplazo.
- **split(String regex), split(String regex, int limit):** Divide esta cadena alrededor de las coincidencias con la expresión regular dada.
- **startsWith(String prefix), startsWith(String prefix, int toffset):** Prueba si esta cadena comienza con el prefijo especificado.

- **strip(), stripLeading(), stripTrailing():** Devuelve una cadena cuyos valores son los de esta cadena, con todos los espacios en blanco liderantes y/o finales eliminados.
- **subSequence(int beginIndex, int endIndex):** Devuelve una secuencia de caracteres que es una subsecuencia de esta secuencia.
- **substring(int beginIndex), substring(int beginIndex, int endIndex):** Devuelve una nueva cadena que es una subcadena de esta cadena.
- **toCharArray():** Convierte esta cadena en una nueva matriz de caracteres.
- **toLowerCase(), toLowerCase(Locale locale):** Convierte todos los caracteres de esta cadena a minúsculas utilizando las reglas del local predeterminado o del especificado.
- **toUpperCase(), toUpperCase(Locale locale):** Convierte todos los caracteres de esta cadena a mayúsculas utilizando las reglas del local predeterminado o del especificado.
- **translateEscapes():** Devuelve una cadena cuyo valor es el de esta cadena, con cualquier secuencia de escape traducida.
- **trim():** Devuelve una copia de la cadena, con los espacios en blanco iniciales y finales omitidos.

Anexo B

En este anexo, puedes ver los distintos métodos que tiene la **clase Math** explicados brevemente:

- **E:** Constante matemática e, la base del logaritmo natural. (Tipo de dato: double)
- **PI:** Constante matemática π, la relación entre la circunferencia de un círculo y su diámetro. (Tipo de dato: double)
- **IEEEremainder(double dividendo, double divisor):** Calcula el residuo de la división de dos números en formato de punto flotante según la norma IEEE 754. (Tipo de dato: double)
- **abs(int valor):** Devuelve el valor absoluto de un número entero. (Tipo de dato: int)

- **absExact(int valor):** Devuelve el valor absoluto de un número entero, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **acos(double angulo):** Devuelve el arcocoseno de un número en formato de punto flotante. (Tipo de dato: double)
- **addExact(int a, int b):** Devuelve la suma de dos números enteros, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **asin(double angulo):** Devuelve el arcoseno de un número en formato de punto flotante. (Tipo de dato: double)
- **atan(double angulo):** Devuelve el arcotangente de un número en formato de punto flotante. (Tipo de dato: double)
- **atan2(double y, double x):** Devuelve el arcotangente del cociente de dos números en formato de punto flotante. (Tipo de dato: double)
- **cbrt(double valor):** Devuelve la raíz cúbica de un número en formato de punto flotante. (Tipo de dato: double)
- **ceil(double valor):** Redondea hacia arriba un número en formato de punto flotante al entero más cercano. (Tipo de dato: double)
- **class:** Devuelve el objeto Class que representa la clase de la instancia actual. (Tipo de dato: Class<?>)
- **copySign(double magnitud, double signo):** Devuelve un número con la magnitud del primer parámetro y el signo del segundo parámetro. (Tipo de dato: double)
- **cos(double angulo):** Devuelve el coseno de un ángulo en formato de punto flotante. (Tipo de dato: double)
- **cosh(double angulo):** Devuelve el coseno hiperbólico de un número en formato de punto flotante. (Tipo de dato: double)
- **decrementExact(int valor):** Resta 1 a un número entero, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **exp(double valor):** Devuelve el valor de “e” elevado a la potencia de un número en formato de punto flotante. (Tipo de dato: double)
- **expm1(double valor):** Devuelve el valor de “e” elevado a la potencia de un número en formato de punto flotante, menos 1. (Tipo de dato: double)

- **floor(double valor):** Redondea hacia abajo un número en formato de punto flotante al entero más cercano. (Tipo de dato: double)
- **floorDiv(int dividendo, int divisor):** Realiza la división entera de dos números enteros y devuelve el resultado redondeado hacia abajo. (Tipo de dato: int)
- **floorMod(int dividendo, int divisor):** Calcula el módulo de dos números enteros y devuelve el resultado redondeado hacia abajo. (Tipo de dato: int)
- **fma(double a, double b, double c):** Realiza una multiplicación y suma precisa de tres números en formato de punto flotante. (Tipo de dato: double)
- **getExponent(double valor):** Devuelve el exponente de un número en formato de punto flotante. (Tipo de dato: int)
- **hypot(double cateto1, double cateto2):** Calcula la hipotenusa de un triángulo rectángulo dados los dos catetos. (Tipo de dato: double)
- **incrementExact(int valor):** Suma 1 a un número entero, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **log(double valor):** Calcula el logaritmo natural de un número en formato de punto flotante. (Tipo de dato: double)
- **log10(double valor):** Calcula el logaritmo en base 10 de un número en formato de punto flotante. (Tipo de dato: double)
- **log1p(double valor):** Calcula el logaritmo natural de (1 + valor) de forma precisa para valores cercanos a 0. (Tipo de dato: double)
- **max(int a, int b):** Devuelve el valor máximo entre dos números enteros. (Tipo de dato: int)
- **min(int a, int b):** Devuelve el valor mínimo entre dos números enteros. (Tipo de dato: int)
- **multiplyExact(int a, int b):** Multiplica dos números enteros, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **multiplyFull(int a, int b):** Multiplica dos números enteros y devuelve los 64 bits de más peso del resultado. (Tipo de dato: long)
- **multiplyHigh(int a, int b):** Multiplica dos números enteros y devuelve los 32 bits de más peso del resultado. (Tipo de dato: int)

- **negateExact(int valor):** Cambia el signo de un número entero, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **nextAfter(double start, double direction):** Devuelve el número más cercano al primer parámetro en la dirección del segundo parámetro. (Tipo de dato: double)
- **nextDown(double valor):** Devuelve el número más cercano al valor especificado, menor que dicho valor. (Tipo de dato: double)
- **nextUp(double valor):** Devuelve el número más cercano al valor especificado, mayor que dicho valor. (Tipo de dato: double)
- **pow(double base, double exponente):** Calcula la potencia de un número en formato de punto flotante. (Tipo de dato: double)
- **random():** Genera un número pseudoaleatorio en el rango de 0.0 (incluido) a 1.0 (excluido). (Tipo de dato: double)
- **rint(double valor):** Redondea un número en formato de punto flotante al entero más cercano. (Tipo de dato: double)
- **round(float valor):** Redondea un número en formato de punto flotante al entero más cercano. (Tipo de dato: int)
- **round(double valor):** Redondea un número en formato de punto flotante al entero más cercano. (Tipo de dato: long)
- **scalb(double valor, int escala):** Escala un número en formato de punto flotante según una potencia de 2 especificada. (Tipo de dato: double)
- **signum(double valor):** Devuelve el signo de un número en formato de punto flotante. (Tipo de dato: double)
- **sin(double angulo):** Devuelve el seno de un ángulo en formato de punto flotante. (Tipo de dato: double)
- **sinh(double angulo):** Devuelve el seno hiperbólico de un número en formato de punto flotante. (Tipo de dato: double)
- **sqrt(double valor):** Calcula la raíz cuadrada de un número en formato de punto flotante. (Tipo de dato: double)
- **subtractExact(int a, int b):** Resta dos números enteros, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **tan(double angulo):** Devuelve la tangente de un ángulo en formato de punto flotante. (Tipo de dato: double)

- **tanh(double angulo):** Devuelve la tangente hiperbólica de un número en formato de punto flotante. (Tipo de dato: double)
- **toDegrees(double anguloRad):** Convierte un ángulo en radianes a grados. (Tipo de dato: double)
- **toIntExact(long valor):** Convierte un número largo a un entero, lanzando una excepción si el resultado no se puede representar como un entero. (Tipo de dato: int)
- **toRadians(double anguloDeg):** Convierte un ángulo en grados a radianes. (Tipo de dato: double)
- **ulp(double valor):** Devuelve la unidad de menor magnitud en el rango del valor especificado. (Tipo de dato: double)
