JAVA

Autor:

Kevin Cárdenas.

2023

Índice

1. Java

Java es un lenguaje de programación orientado a objetos diseñado para ser portable, seguro y fácil de entender. La programación orientada a objetos (POO) es una técnica de programación que se centra en la organización del código en torno a objetos, que son entidades que contienen datos y métodos para manipular esos datos.

Java es un lenguaje compilado e interpretado. Cuando se escribe un programa en Java, se escribe en un archivo fuente que luego se compila en un archivo ejecutable. Este archivo ejecutable se puede ejecutar en cualquier plataforma que tenga una máquina virtual Java (JVM) instalada. La JVM es un software que permite que los programas Java se ejecuten en diferentes sistemas operativos sin necesidad de recompilar el código fuente.

El proceso de compilación de Java convierte el código fuente en un formato binario conocido como bytecode, que es un conjunto de instrucciones que la JVM puede interpretar y ejecutar en tiempo de ejecución. La JVM interpreta el bytecode y lo convierte en código de máquina nativo para la plataforma específica en la que se está ejecutando.

En resumen, Java es un lenguaje orientado a objetos que se compila en bytecode y se interpreta en tiempo de ejecución por la JVM, lo que lo hace altamente portable y seguro.

1.1. Instalación

Te proporciono las instrucciones para instalar Java en Linux utilizando la terminal:

1. Abre una terminal: Abre una ventana de terminal en Linux. En la mayoría de las distribuciones de Linux, puedes abrir una terminal haciendo clic en el botón de aplicaciones en la barra de tareas y buscando "terminal".

2. Actualiza el sistema: Antes de instalar Java, asegúrate de actualizar el sistema. Para hacerlo, ejecuta el siguiente comando en la terminal:

```
sudo apt-get update
```

   Este comando actualizará la lista de paquetes disponibles en el sistema.

3. Instala Java: Para instalar Java en Linux, puedes utilizar el comando apt-get. Ejecuta el siguiente comando en la terminal para instalar la versión más reciente de Java:

```
sudo apt-get install default-jdk
```

Si deseas instalar una versión específica de Java, puedes reemplazar “default-jdk” con el nombre de la versión que deseas instalar (por ejemplo, “openjdk-11-jdk”)

4. Verifica la instalación: Una vez que se haya completado la instalación, verifica que Java se haya instalado correctamente. Para hacerlo, ejecuta el siguiente comando en la terminal:

```
java -version
```

Si deseas instalar inteligent IDEA sólo ejecuta el comando `sudo snap install intellij-idea-community --classic`

Y ahora puedes escribir tu primer “Hola Mundo”, abriendo un nuevo proyecto en nuestro IDE Creando un archivo llamado “HolaMundo.java” y escribiendo en el lo siguiente:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hola Mundo!");
    }
}
```

comienza con la definición de la clase "HelloWorld". Todas las clases de Java deben tener una definición como esta. La clase principal del programa debe tener el mismo nombre que el archivo de código fuente (en este caso, “HolaMundo.java”).

Dentro de la clase “HelloWorld”, se define un método llamado “main”. Este método es el punto de entrada del programa. Todos los programas de Java deben tener un método “main” como este.

Dentro del método “main”, se utiliza la función “println” de la clase “System” para imprimir el mensaje “Hola Mundo!” en la consola. El mensaje debe estar entre comillas dobles para indicar que es una cadena de texto.

1.2. Tipos de Variables

En Java, existen diferentes tipos de variables que se pueden utilizar para almacenar diferentes tipos de datos. Algunos de los tipos de variables más comunes en Java son:

- **int**: este tipo de variable se utiliza para almacenar valores enteros. Por ejemplo: `int edad = 30;`
- **double**: este tipo de variable se utiliza para almacenar valores decimales. Por ejemplo: `double altura = 1.75;`
- **boolean**: este tipo de variable se utiliza para almacenar valores booleanos (verdadero o falso). Por ejemplo: `boolean esMayorDeEdad = true;`

- **char**: este tipo de variable se utiliza para almacenar caracteres individuales. Por ejemplo: `char inicial = 'J';`
- **String**: este tipo de variable se utiliza para almacenar cadenas de texto. A diferencia de los otros tipos, la sintaxis de una variable de tipo String se escribe con mayúscula al inicio. Por ejemplo: `String nombre = "Juan";`

Además de estos tipos de variables básicos, existen otros tipos más complejos en Java, como arreglos, listas, objetos, etc.

Es importante tener en cuenta que las variables en Java son de tipado estático, lo que significa que una vez que se ha declarado el tipo de una variable, no se puede cambiar. Esto ayuda a prevenir errores comunes en tiempo de ejecución.

Por ejemplo:

```
public class EjemploVariables {

    public static void main(String[] args) {

        int edad = 30;
        double altura = 1.75;
        boolean esMayorDeEdad = true;
        char inicial = 'J';
        String nombre = "Juan";

        System.out.println("Nombre: " + nombre);
        System.out.println("Edad: " + edad);
        System.out.println("Altura: " + altura);
        System.out.println("Es mayor de edad?: " + esMayorDeEdad);
        System.out.println("Inicial: " + inicial);
    }
```

En este ejemplo, se declaran variables de diferentes tipos y se les asignan valores. Luego, se imprimen los valores de estas variables en la consola utilizando el método "println" de la clase "System". La salida del programa sería la siguiente:

```
Nombre: Juan
Edad: 30
Altura: 1.75
Es mayor de edad?: true
Inicial: J
```

En este caso, el programa imprime los valores de las variables “nombre”, “edad”, “altura”, “esMayorDeEdad” e “inicial”. Cada valor se concatena con una cadena de texto que describe qué valor se está imprimiendo.

En programación orientada a objetos (POO), además de los tipos de variables primitivos como `int`, `double`, `boolean` y `char`, existen también los tipos de variables de objeto, que se definen a partir de clases.

Los tipos de variables de objeto se utilizan para representar objetos de una clase. Por ejemplo, si tenemos una clase `Persona`, podemos crear un objeto `persona1` de esa clase y utilizar una variable para hacer referencia a ese objeto:

```
Persona persona1 = new Persona();
```

En este caso, `persona1` es una variable de tipo `Persona` que hace referencia a un objeto de la clase `Persona`. Podemos utilizar métodos y propiedades de la clase `Persona` para interactuar con el objeto al que hace referencia la variable `persona1`.

Además, en POO también existe el concepto de variables de instancia y variables de clase. Las variables de instancia son variables que se definen en una clase y que pertenecen a cada objeto creado a partir de esa clase. Por ejemplo, si tenemos una clase `Persona` con una variable de instancia `nombre`, cada objeto `persona1`, `persona2`, etc. tendría su propio valor de la variable `nombre`.

Por otro lado, las variables de clase se definen en una clase y son compartidas por todos los objetos creados a partir de esa clase. Para definir una variable de clase, se utiliza la palabra clave `static`. Por ejemplo, si queremos tener una variable `contador` que cuente la cantidad de objetos `Persona` creados, podemos definirla como una variable de clase:

```
public class Persona {
    static int contador = 0;
    String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
        contador++;
    }
}
```

En este caso, la variable `contador` se incrementa cada vez que se crea un objeto `Persona`. Todas las instancias de la clase `Persona` comparten la misma variable `contador`.

Longitud de un número

El tipo de dato “int” en Java es un tipo primitivo que representa números enteros. Este tipo de dato es utilizado para almacenar valores numéricos que no tienen decimales, y ocupa 32 bits (4 bytes) en la

memoria.

Esto significa que un valor de tipo “int” puede representar cualquier número entero en el rango de “-2,147,483,648” a “2,147,483,647”. Si intentamos almacenar un valor fuera de este rango, se producirá un error en tiempo de ejecución.

Es importante tener en cuenta que el tipo de dato “int” es de tamaño fijo, lo que significa que siempre ocupa la misma cantidad de memoria, independientemente del valor que esté almacenando. Esto lo hace más eficiente en términos de memoria que otros tipos de datos que permiten una mayor precisión, como el tipo “double” o “float”.

Si necesitamos almacenar números enteros fuera del rango de un “int”, podemos utilizar tipos de datos de mayor tamaño, como “long”, que ocupa 64 bits (8 bytes) en la memoria y puede almacenar valores en el rango de “-9,223,372,036,854,775,808” a “9,223,372,036,854,775,807”.

Diferencia entre float y double

En Java, existen dos tipos de datos de punto flotante: **float** y **double**. Ambos se utilizan para representar números decimales, pero difieren en su precisión y tamaño en memoria.

El tipo de dato **float** ocupa 32 bits en memoria y tiene una precisión de aproximadamente 6-7 dígitos decimales significativos. Por otro lado, el tipo de dato **double** ocupa 64 bits en memoria y tiene una precisión de aproximadamente 15-16 dígitos decimales significativos.

Por lo tanto, se recomienda utilizar el tipo de dato **float** cuando se necesita una precisión menor y se quiere ahorrar memoria. Por ejemplo, cuando se trabaja con grandes cantidades de datos en una aplicación. Por otro lado, se recomienda utilizar el tipo de dato **double** cuando se necesita una mayor precisión y no se tiene restricción en cuanto al uso de memoria.

Es importante tener en cuenta que al asignar valores a variables de tipo **float** o **double**, se deben incluir los sufijos **f** o **d**, respectivamente. Por ejemplo:

```
1 float numeroFloat = 3.14159f;
2 double numeroDouble = 3.14159d;
```

De lo contrario, el compilador de Java asumirá que se trata de un valor de tipo **double**.

Tipo de dato char

El tipo de dato `char` en Java es utilizado para almacenar un único carácter alfanumérico o un símbolo en Unicode. La sintaxis para declarar una variable de tipo `char` es la siguiente:

```
1 char variableChar = 'a';
```

Aquí, se declara una variable llamada `variableChar` de tipo `char`, y se le asigna el valor de la letra 'a'. Es importante notar que los valores de tipo `char` siempre deben estar encerrados en comillas simples.

Además, el tipo de dato `char` también puede representar valores numéricos utilizando su representación en la tabla ASCII. Por ejemplo:

```
char variableNum = 65;
```

Aquí, la variable `variableNum` de tipo `char` representa el valor numérico 65 en la tabla ASCII, que es la letra 'A'. Es importante mencionar que la representación en ASCII está limitada a los caracteres del conjunto ASCII, por lo que algunos caracteres especiales pueden no tener una representación ASCII válida.

Es importante tener en cuenta que una variable `char` ocupa 2 bytes de memoria en Java. Esto significa que puede representar valores de caracteres Unicode que se encuentran en el rango de 0 a 65,535.

Tipo de dato boolean

El tipo de dato **boolean** en Java es un tipo de dato que puede tener dos valores: **true** o **false**. Este tipo de dato es útil cuando se desea representar una condición lógica en el programa.

A diferencia de otros tipos de datos como int o double, que pueden tener una amplia gama de valores, un boolean solo puede tener dos posibles valores. Estos valores son útiles para representar situaciones donde una condición es verdadera o falsa, como por ejemplo en una sentencia de control de flujo como un **if** o un **while**.

Es importante recordar que los valores true y false en Java son sensibles a mayúsculas y minúsculas. Además, el tipo de dato boolean solo ocupa un bit en la memoria, lo que lo convierte en uno de los tipos de datos más eficientes en términos de espacio de almacenamiento.

Tipo de dato var

A partir de Java 10, se introdujo el tipo de dato `var`, que permite al compilador inferir el tipo de la variable en tiempo de compilación.

En lugar de declarar una variable como:

```
String nombre = "Juan";
```

Podemos usar `var`:

```
var nombre = "Juan";
```

En este caso, el compilador inferirá que el tipo de la variable `nombre` es `String`. También podemos usar `var` para tipos de datos más complejos, como colecciones:

```java
var lista = new ArrayList<String>();
```

En este caso, el compilador infiere que el tipo de la variable `lista` es `ArrayList<String>`. Sin embargo, es importante tener en cuenta que el tipo de la variable no es dinámico, una vez que se ha inferido en tiempo de compilación, el tipo de la variable no puede cambiar.

El uso de `var` puede hacer que el código sea más conciso y fácil de leer, especialmente en casos donde el tipo de dato es complejo o largo de escribir. Sin embargo, también puede hacer que el código sea menos legible si se abusa de su uso o si no se usa con cuidado.

Variables constantes

En Java, también se pueden declarar variables que no pueden cambiar su valor después de haber sido inicializadas. Estas variables se llaman **variables constantes** o **constantes** y se declaran utilizando la palabra clave `final`. Por convención, el nombre de las constantes se escribe en mayúsculas y se separan las palabras con guiones bajos (`CONSTANTE_EJEMPLO`). Por ejemplo, podemos declarar una constante para el valor de PI de la siguiente manera:

1.2.1. Arrays en Java

Un arreglo (**array**) en Java es una estructura de datos que permite almacenar un conjunto de elementos del mismo tipo. Los elementos de un array se almacenan en posiciones consecutivas de memoria y se acceden mediante un índice numérico.

Para declarar un array en Java, se debe especificar el tipo de datos de los elementos y la cantidad de elementos que tendrá el array. Por ejemplo, para declarar un array de enteros con 5 elementos, se utilizaría la siguiente sintaxis:

```java
int[] miArray = new int[5];
```

Es importante tener en cuenta que los índices en un array comienzan en cero y terminan en el tamaño del array menos uno. Por ejemplo, si se tiene un array de enteros con 5 elementos, el primer elemento se accedería con el índice 0 y el último con el índice 4.

Además de la declaración básica de un array, existen varios métodos útiles para manipularlos en Java:

- **length**: devuelve la longitud del array.
- **clone**: crea una copia exacta del array.
- **sort**: ordena los elementos del array en orden ascendente.

También es posible declarar arrays multidimensionales en Java, es decir, arrays que contienen otros arrays como elementos. Por ejemplo, se podría declarar un array bidimensional de enteros de la siguiente manera:

```
int[][] miArrayBidimensional = new int[3][2];
```

Este array tendría 3 elementos, cada uno de los cuales sería un array de enteros con 2 elementos. Para acceder a un elemento específico de un array multidimensional, se deben utilizar índices separados por comas. Por ejemplo, para acceder al segundo elemento del primer array del array bidimensional anterior, se utilizaría la siguiente sintaxis:

```
int segundoElemento = miArrayBidimensional[0][1];
```

Es importante tener en cuenta que los arrays en Java tienen un tamaño fijo una vez que se han creado. Si se necesita una estructura de datos dinámica que permita agregar o eliminar elementos, se recomienda utilizar las colecciones de Java, como por ejemplo ArrayList.

Índices y búsqueda de elementos en Arrays

Los elementos de un array en Java se pueden acceder mediante su índice. El índice de un elemento en un array es un número entero que indica su posición dentro del array. En Java, los índices comienzan en cero, lo que significa que el primer elemento de un array tiene un índice de cero, el segundo elemento tiene un índice de uno, y así sucesivamente.

Para acceder a un elemento en un array, se utiliza la sintaxis de corchetes []. El índice del elemento deseado se coloca entre los corchetes. Por ejemplo, si queremos acceder al tercer elemento de un array llamado "numeros", se utiliza la siguiente sintaxis:

```
int[] numeros = {1, 2, 3, 4, 5};
int tercerNumero = numeros[2]; // Tercer elemento, indice 2
```

En este ejemplo, el tercer elemento del array “numeros” tiene un índice de 2, por lo que se utiliza la sintaxis “numeros[2]” para acceder a ese elemento.

Para buscar un elemento en un array, se puede utilizar un ciclo “for” para iterar a través de todos los elementos y comparar cada uno con el elemento deseado. Sin embargo, Java también proporciona un método llamado “binarySearch()” que puede buscar un elemento en un array ordenado de manera eficiente.

El método “binarySearch()” utiliza el algoritmo de búsqueda binaria para encontrar el elemento deseado en el array. El algoritmo de búsqueda binaria funciona dividiendo el array en dos mitades y comparando el elemento deseado con el elemento en el medio. Si el elemento deseado es menor que el elemento en el medio, se busca en la mitad inferior del array. Si es mayor, se busca en la mitad superior del array.

Este proceso se repite hasta que se encuentra el elemento deseado o se determina que no está presente en el array.

El método "binarySearch()" tiene la siguiente sintaxis:

```
Arrays.binarySearch(array, elemento);
```

Donde "array" es el array en el que se desea buscar el elemento, y "elemento" es el elemento que se desea buscar. El método devuelve el índice del elemento si se encuentra en el array, o un número negativo que indica el punto de inserción si el elemento no está presente en el array.

Además del método "binarySearch()", Java proporciona otros métodos útiles para trabajar con arrays, como "sort()", que ordena los elementos de un array en orden ascendente, y "equals()", que compara dos arrays para determinar si son iguales.

1.2.2. Uso de variables

Para utilizar una variable en Java, primero se debe declarar, lo que significa reservar un espacio en memoria para la variable. La declaración de una variable se realiza especificando el tipo y el nombre de la variable, como se muestra en el siguiente ejemplo:

```
int edad;
```

En este caso, se está declarando una variable de tipo `int` llamada `edad`. Después de la declaración, se puede asignar un valor a la variable utilizando el operador de asignación `=`, como se muestra a continuación:

```
edad = 25;
```

En este caso, se está asignando el valor `25` a la variable `edad`.

También se puede declarar y asignar un valor a una variable en la misma línea, como se muestra en el siguiente ejemplo:

```
double precio = 10.99;
```

En este caso, se está declarando una variable de tipo `double` llamada `precio` y se le está asignando el valor `10.99`.

Una vez que se ha declarado y asignado un valor a una variable, se puede utilizar en el código para hacer cálculos y tomar decisiones. Por ejemplo:

```
int edad = 25;
if (edad >= 18) {
System.out.println("Eres mayor de edad");
} else {
System.out.println("Eres menor de edad");
```

```
}
```

En este caso, se está utilizando la variable `edad` para determinar si una persona es mayor o menor de edad.

También se puede actualizar el valor de una variable utilizando el operador de asignación. Por ejemplo:

```
int contador = 0;
contador = contador + 1;
```

En este caso, se está incrementando el valor de la variable `contador` en uno.

Es importante tener en cuenta que las variables en Java ocupan espacio en memoria y que este espacio es limitado. Por lo tanto, es importante utilizar variables de manera eficiente e intentar no crear más variables de las necesarias. Además, las variables que ya no se necesitan podriamos eliminarlas para liberar espacio en memoria y evitar fugas de memoria.

1.2.3. Convención de nombres

La convención de nombres en programación es importante para escribir código legible y fácilmente comprensible. En Java, existen algunas reglas de convención de nombres que se deben seguir para escribir código legible y coherente.

Primero, es importante recordar que Java es sensible a mayúsculas y minúsculas, lo que significa que una letra mayúscula se considera diferente de su contraparte en minúscula. Por ejemplo, la variable "nombre" es diferente de la variable "Nombre" en Java.

Además, existen algunas convenciones de nomenclatura comunes que se utilizan en Java. Por ejemplo, las variables generalmente se nombran usando una convención llamada "camelCase". En camelCase, el primer carácter de la primera palabra es en minúscula, y la primera letra de cada palabra subsiguiente se escribe en mayúscula. Por ejemplo, "nombreDeUsuario" es un buen nombre de variable en camelCase.

En cuanto a los caracteres permitidos en los nombres de variables en Java, estos pueden contener letras, números y guiones bajos. Sin embargo, el primer carácter de un nombre de variable no puede ser un número. Además, los nombres de variables no pueden incluir espacios ni caracteres especiales como signos de puntuación o símbolos matemáticos.

Es importante seguir estas convenciones de nomenclatura para que el código sea fácil de leer y entender. Además, seguir las convenciones de nomenclatura también puede ayudar a prevenir errores en el código y hacer que sea más fácil de mantener y actualizar en el futuro.

El libro "Clean Code" de Robert C. Martin es una gran referencia para escribir código limpio y legible. Aquí hay un resumen de algunos de los conceptos clave:

- Nombres descriptivos: Elige nombres que describan claramente lo que hace una variable, método o clase. Nombres descriptivos hacen que el código sea más fácil de entender y de mantener.
- Funciones y métodos cortos: Las funciones y los métodos deben ser lo más cortos posible, idealmente menos de 20 líneas de código. Esto hace que el código sea más fácil de entender y de probar.
- Mantener un estilo consistente: Es importante mantener un estilo consistente en todo el código, desde la indentación hasta la nomenclatura de las variables. Esto hace que el código sea más fácil de leer y de mantener.
- Eliminar duplicación: La duplicación de código puede ser un problema importante en el código, ya que hace que el código sea más difícil de mantener y actualizar. Es importante identificar y eliminar la duplicación siempre que sea posible.
- Comentarios: Los comentarios deben ser utilizados para explicar el por qué del código, no el qué. El código debe ser lo suficientemente claro para entender lo que está haciendo sin necesidad de comentarios. Los comentarios pueden ser útiles para explicar decisiones arquitectónicas o decisiones de diseño.
- Pruebas unitarias: Las pruebas unitarias son esenciales para escribir código limpio y de calidad. Las pruebas unitarias aseguran que el código funciona correctamente y que los cambios en el código no introducen nuevos errores.

Estos son sólo algunos de los conceptos clave que se discuten en "Clean Code". Siguiendo estos principios, podemos escribir código limpio y fácil de entender que es más fácil de mantener y actualizar en el futuro.

Convención de nombres: Upper Camel Case y Lower Camel Case

En Java, se utilizan dos convenciones de nombres comunes: Upper Camel Case y Lower Camel Case.

Upper Camel Case, también conocido como **PascalCase**, se utiliza para nombrar clases y tipos de datos. En Upper Camel Case, la primera letra de cada palabra se escribe en mayúscula, y no se utilizan espacios ni guiones bajos para separar las palabras. Por ejemplo:

```
1 public class MiClaseEjemplo {
2 // class code
3 }
```

Lower Camel Case, también conocido como **camelCase**, se utiliza para nombrar variables, métodos y funciones. En Lower Camel Case, la primera letra de la primera palabra se escribe en minúscula, y la primera letra de cada palabra siguiente se escribe en mayúscula. No se utilizan espacios ni guiones bajos para separar las palabras. Por ejemplo:

```
int edadDelUsuario = 25;
public void miMetodoEjemplo() {
// method code
}
```

1.3. Operadores

En programación, los operadores son símbolos o palabras reservadas que se utilizan para realizar operaciones matemáticas o lógicas sobre variables o valores. En Java, existen diferentes tipos de operadores que se utilizan en distintas situaciones.

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas entre variables numéricas. Los operadores aritméticos en Java son los siguientes:

- Suma (+): se utiliza para sumar dos valores.
- Resta (-): se utiliza para restar un valor de otro.
- Multiplicación (*): se utiliza para multiplicar dos valores.
- División (/): se utiliza para dividir un valor por otro.
- Módulo (%): se utiliza para obtener el resto de una división.

Operadores de asignación

Los operadores de asignación se utilizan para asignar un valor a una variable. El operador de asignación en Java es el signo igual (=). También existen operadores de asignación compuestos, que realizan una operación y luego asignan el resultado a la variable. Algunos ejemplos son:

- +=: suma el valor de la variable y el valor especificado y asigna el resultado a la variable.
- -=: resta el valor especificado de la variable y asigna el resultado a la variable.
- *=: multiplica el valor de la variable y el valor especificado y asigna el resultado a la variable.
- /=: divide el valor de la variable por el valor especificado y asigna el resultado a la variable.
- %=: obtiene el resto de la división entre la variable y el valor especificado y asigna el resultado a la variable.

Operadores de comparación

Los operadores de comparación se utilizan para comparar dos valores y devuelven un valor booleano (verdadero o falso) como resultado. Los operadores de comparación en Java son los siguientes:

- Igual que (==): devuelve verdadero si los dos valores son iguales.
- Distinto que (!=): devuelve verdadero si los dos valores son distintos.
- Mayor que (>): devuelve verdadero si el primer valor es mayor que el segundo valor.
- Menor que (<): devuelve verdadero si el primer valor es menor que el segundo valor.
- Mayor o igual que (>=): devuelve verdadero si el primer valor es mayor o igual que el segundo valor.
- Menor o igual que (<=): devuelve verdadero si el primer valor es menor o igual que el segundo valor.

Operadores lógicos

Los operadores lógicos son aquellos que se utilizan para realizar operaciones booleanas entre dos expresiones, cuyo resultado será verdadero o falso. Los operadores lógicos en Java son los siguientes:

- **AND lógico (&&)**: Este operador devuelve verdadero si ambas expresiones son verdaderas.
- **OR lógico (||)**: Este operador devuelve verdadero si al menos una de las expresiones es verdadera.
- **NOT lógico (!)**: Este operador se utiliza para negar el resultado de una expresión booleana. Si la expresión original es verdadera, el resultado será falso, y viceversa.

A continuación, se muestran algunos ejemplos de uso de operadores lógicos:

```
boolean a = true;
boolean b = false;

boolean c = a && b; // false

boolean d = a || b; // true

boolean e = !a; // false
```

Podemos hacer un ejemplo sencillo que combine varios conceptos que hemos visto. Por ejemplo, podemos escribir un programa que le pregunte al usuario su edad y, dependiendo de si es mayor o menor de edad, le dé la bienvenida o le indique que debe esperar un poco más para poder acceder al contenido.

Vamos a crear un programa que permita al usuario ingresar su edad y verifique si es mayor de edad o no. Para ello, utilizaremos los conceptos vistos anteriormente.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Ingrese su edad: ");
    int edad = input.nextInt();

    boolean esMayorDeEdad = edad >= 18;

    System.out.println("Usted tiene " + edad + " a~nos.");
    System.out.println("Es mayor de edad?: " + esMayorDeEdad);
}
}
```

En este programa, utilizamos la clase `Scanner` para leer la entrada del usuario desde la consola. Luego, guardamos la edad ingresada en una variable de tipo `int` llamada `edad`.

Después, utilizamos una variable de tipo `boolean` llamada `esMayorDeEdad` para verificar si la edad ingresada es mayor o igual a 18 años.

Finalmente, imprimimos la edad ingresada y si el usuario es mayor de edad o no utilizando la función `println` de la clase `System` y concatenando las variables correspondientes.

1.3.1. Operaciones matemáticas con Math

En Java, la clase Math proporciona una serie de métodos estáticos para realizar operaciones matemáticas. A continuación se muestran algunos ejemplos:

- "**max**": El método "max" de Math toma dos números como argumentos y devuelve el mayor de los dos. Por ejemplo:

```
    int a = 10;
    int b = 20;
    int maximo = Math.max(a, b); // devuelve 20

```

- "**min**": El método "min" de Math toma dos números como argumentos y devuelve el menor de los dos. Por ejemplo:

```
int a = 10;
int b = 20;
int minimo = Math.min(a, b); // devuelve 10
```

- "**abs**": El método "abs" de Math devuelve el valor absoluto de un número. Por ejemplo:

```
int a = -10;
int valorAbsoluto = Math.abs(a); // devuelve 10
```

- "**sqrt**": El método "sqrt" de Math devuelve la raíz cuadrada de un número. Por ejemplo:

```
int a = 25;
double raizCuadrada = Math.sqrt(a); // devuelve 5.0
```

- "**pow**": El método "pow" de Math toma dos números como argumentos y devuelve el resultado de elevar el primer número a la potencia del segundo número. Por ejemplo:

```
int base = 2;
int exponente = 3;
double potencia = Math.pow(base, exponente); // devuelve 8.0
```

- "**sin, cos, tan**": Los métodos "sin", "cos" y "tan" de Math calculan el seno, coseno y tangente de un ángulo en radianes, respectivamente. Por ejemplo:

```
double angulo = Math.PI / 4; // angulo de 45 grados en radianes
double seno = Math.sin(angulo); // devuelve 0.7071067811865475
double coseno = Math.cos(angulo); // devuelve 0.7071067811865476
double tangente = Math.tan(angulo); // devuelve 0.9999999999999999
```

Es importante tener en cuenta que los métodos trigonométricos de Math toman como argumento un ángulo en radianes, no en grados. Para convertir de grados a radianes, se puede utilizar la siguiente fórmula:

$$\theta_{rad} = \frac{\theta_{grados} \times \pi}{180}$$

donde θ_{rad} es el ángulo en radianes y θ_{grados} es el ángulo en grados.

Por ejemplo, si se quisiera calcular el seno de 45 grados utilizando el método sin() de Math, se debería convertir primero a radianes utilizando la fórmula anterior:

$$45_{grados} = \frac{\pi}{4}_{radianes}$$

Entonces, el código para calcular el seno de 45 grados utilizando el método sin() sería el siguiente:

```java
double anguloEnGrados = 45;
double anguloEnRadianes = anguloEnGrados * Math.PI / 180;
double seno = Math.sin(anguloEnRadianes);
System.out.println("El seno de " + anguloEnGrados + " grados es: " + seno);
```

El resultado sería: `El seno de 45.0 grados es: 0.7071067811865475`

1.4. Casting en Variables

El casting en Java es la conversión explícita de una variable de un tipo de dato a otro. Esto se hace poniendo entre paréntesis el tipo de dato al que se desea convertir la variable, seguido del nombre de la variable.

1.4.1. Casting Manual

En Java, algunos tipos de datos se pueden convertir manualmente en otros tipos de datos. Un ejemplo común es la conversión de un `double` a un `int`. Para hacer esto, simplemente se escribe el tipo de dato al que se desea convertir la variable entre paréntesis, seguido del nombre de la variable:

```java
double d = 10.5;
int i = (int) d;
System.out.println(i); // salida: 10
```

También se puede hacer lo contrario, es decir, convertir un `int` a un `double`:

```java
int i = 10;
double d = (double) i;
System.out.println(d); // salida: 10.0
```

1.4.2. Casting Automático

En algunos casos, Java realiza la conversión de un tipo de dato a otro de forma automática. Esto se conoce como casting automático y sucede cuando se asigna una variable de un tipo de dato a una variable de otro tipo de dato compatible. Por ejemplo, cuando se asigna una variable `int` a una variable `double`, Java convierte automáticamente el `int` a un `double`:

```java
int i = 10;
double d = i;
System.out.println(d); // salida: 10.0
```

Sin embargo, no todos los tipos de datos son compatibles entre sí. En estos casos, es necesario hacer un casting manual para convertir la variable.

1.4.3. Casting en otros tipos de datos

El casting manual también se puede utilizar en otros tipos de datos, como en el caso de los tipos de datos `char` y `boolean`. Por ejemplo, para convertir un `char` a un `int`, se puede hacer lo siguiente:

```
char c = 'a';
int i = (int) c;
System.out.println(i); // salida: 97
```

Y para convertir un `boolean` a un `int`, se puede hacer lo siguiente:

```
boolean b = true;
int i = b ? 1 : 0;
System.out.println(i); // salida: 1
```

Es importante tener en cuenta que los tipos de datos deben ser compatibles entre sí para poder hacer un casting automático. En caso contrario, es necesario hacer un casting manual para convertir la variable al tipo de dato deseado.

Versiones de Java y JDK

Java es un lenguaje de programación popular utilizado en una variedad de aplicaciones, desde aplicaciones web hasta aplicaciones móviles. A lo largo de los años, ha habido varias versiones de Java, cada una con sus propias características y mejoras.

El Java Development Kit (JDK) es un kit de herramientas de desarrollo para Java que incluye el compilador Java, la biblioteca de clases Java y otras herramientas de desarrollo. El JDK es necesario para desarrollar y ejecutar programas en Java.

A continuación, se muestran algunas de las versiones de Java y JDK más populares:

Java SE 8

Java SE 8 es una de las versiones más populares de Java. Fue lanzada en marzo de 2014 y es conocida por sus mejoras en el rendimiento y la seguridad. Java SE 8 también introdujo la programación funcional en Java con la adición de las expresiones lambda.

Java SE 11

Java SE 11 es otra versión popular de Java. Fue lanzada en septiembre de 2018 y es conocida por sus mejoras en la seguridad y la estabilidad. Java SE 11 también introdujo la noción de lanzamientos a largo plazo (LTS), que son lanzamientos que reciben soporte a largo plazo.

Java SE 16

Java SE 16 es la versión más reciente de Java en el momento de escribir esto. Fue lanzada en marzo de 2021 y es conocida por sus mejoras en la programación funcional y en la seguridad.

Es importante tener en cuenta que las diferentes versiones de Java pueden tener diferentes funcionalidades y características. Al seleccionar una versión de Java, es importante considerar las necesidades del proyecto y las características específicas de cada versión.

Además, es importante tener en cuenta que la elección del JDK también es importante para el desarrollo de programas en Java. Es recomendable utilizar la última versión del JDK para aprovechar al máximo las características y mejoras de Java.

Archivos .jar

Un archivo JAR (Java ARchive) es un archivo comprimido que contiene código y recursos que puede ser utilizado por una máquina virtual de Java. Estos archivos suelen utilizarse para distribuir bibliotecas de clases reutilizables o aplicaciones completas escritas en Java.

Para crear un archivo .jar desde la línea de comandos, se utiliza el comando `jar` proporcionado por el JDK de Java. La sintaxis básica del comando es la siguiente:

```
jar cf jar-file input-file(s)
```

Donde `jar-file` es el nombre del archivo .jar que se desea crear, y `input-file(s)` es la lista de archivos que se desean incluir en el archivo .jar. Por ejemplo, para crear un archivo .jar que contenga todos los archivos .class en el directorio actual, se puede utilizar el siguiente comando:

```
jar cf mylibrary.jar *.class
```

Para utilizar una biblioteca JAR en un proyecto de Java, primero se debe agregar la biblioteca al classpath del proyecto. Esto se puede hacer de varias formas, dependiendo del entorno de desarrollo utilizado. En Eclipse, por ejemplo, se puede agregar una biblioteca JAR al classpath de un proyecto haciendo clic derecho en el proyecto, seleccionando “Propiedades”, y luego seleccionando “Java Build Path” en el menú de la izquierda. A continuación, se debe hacer clic en la pestaña “Librerías”, y luego en el botón “Agregar JARs...” para seleccionar el archivo .jar que se desea agregar.

Una vez que se ha agregado la biblioteca al classpath del proyecto, se pueden importar las clases y utilizarlas en el código de la misma manera que se importan y utilizan las clases de la biblioteca estándar de Java.

Es importante tener en cuenta que los archivos JAR pueden contener código malicioso o peligroso. Por esta razón, se debe tener precaución al descargar y utilizar archivos JAR de fuentes desconocidas.

1.5. Ciclos en Java

Los ciclos son estructuras de control que permiten repetir una o varias instrucciones mientras se cumpla una condición. En Java, existen varios tipos de ciclos, cada uno con su propia sintaxis y uso específico.

1.5.1. Ciclo condicional: `if`

El ciclo condicional `if` permite ejecutar una o varias instrucciones si se cumple una condición determinada. Su sintaxis es la siguiente:

```
if (condicion) {
// Instrucciones a ejecutar si se cumple la condicion
}
```

Si la condición es verdadera, se ejecutan las instrucciones dentro del bloque de código entre llaves. Si la condición es falsa, se omite la ejecución de las instrucciones y se continúa con la siguiente línea de código. También se puede utilizar la instrucción `else` para especificar un bloque de código a ejecutar en caso de que la condición sea falsa:

```
if (condicion) {
// Instrucciones a ejecutar si se cumple la condicion
} else {
// Instrucciones a ejecutar si NO se cumple la condicion
}
```

1.5.2. `Switch` en Java

En Java, el `switch` es una estructura de control que permite realizar diferentes acciones según el valor de una expresión. La sintaxis básica del `switch` es la siguiente:

```
switch (expresion) {
    case valor1:
        // codigo para valor1
        break;
    case valor2:
        // codigo para valor2
        break;
    case valor3:
        // codigo para valor3
        break;
    default:
        // codigo para cualquier otro valor
```

```
}
```

En este ejemplo, la variable `expresion` es evaluada y se compara con cada uno de los valores especificados en los casos (`valor1`, `valor2` y `valor3`). Si el valor coincide con alguno de los casos, se ejecuta el código correspondiente a ese caso. Si no coincide con ninguno de los casos, se ejecuta el código dentro del bloque `default`.

A continuación se presenta un ejemplo práctico de cómo utilizar el `switch` en Java para calcular el día de la semana a partir de un número del 1 al 7:

```
public class DiaDeLaSemana {
    public static void main(String[] args) {
        int numDia = 3;
        String dia = "";

        switch (numDia) {
            case 1:
                dia = "Lunes";
                break;
            case 2:
                dia = "Martes";
                break;
            case 3:
                dia = "Miercoles";
                break;
            case 4:
                dia = "Jueves";
                break;
            case 5:
                dia = "Viernes";
                break;
            case 6:
                dia = "Sabado";
                break;
            case 7:
                dia = "Domingo";
                break;
            default:
                dia = "Numero invalido";
        }

```

```
        System.out.println("El dia correspondiente al numero " + numDia + " es " + dia + ".");
    }
}
```

En este ejemplo, se utiliza el `switch` para asignar un valor a la variable `dia` según el valor de la variable `numDia`. Luego, se muestra por pantalla el día correspondiente al número ingresado. Si se ingresa un número que no está dentro del rango del 1 al 7, se muestra un mensaje indicando que el número es inválido.

1.5.3. Ciclo repetitivo: `for`

El ciclo for en Java es una estructura de control utilizada para repetir un bloque de código varias veces, mientras se cumpla una condición. En Java, existen dos tipos de ciclo for: el ciclo for normal y el ciclo for each.

Ciclo For Each

El ciclo for each, también conocido como for mejorado, es una forma simplificada de recorrer los elementos de una colección o un arreglo en Java. En lugar de utilizar un índice para acceder a cada elemento, el ciclo for each utiliza una variable para hacer referencia a cada uno de los elementos de la colección.

La sintaxis del ciclo for each en Java es la siguiente:

```
for (tipoDeDato variable : coleccion) {
// Codigo a ejecutar
}
```

En esta sintaxis, “tipoDeDato” es el tipo de dato de la colección, “variable” es la variable que se utilizará para hacer referencia a cada elemento de la colección y “coleccion” es la colección que se va a recorrer.

Por ejemplo, si tenemos un arreglo de números enteros y queremos imprimir cada uno de sus elementos, podemos utilizar un ciclo for each de la siguiente manera:

```
int[] numeros = {1, 2, 3, 4, 5};
for (int numero : numeros) {
System.out.println(numero);
}
```

Este código imprimirá los números 1, 2, 3, 4 y 5 en la consola.

Notación más compacta

En Java, existen algunas formas de hacer la sintaxis del ciclo for más compacta. Por ejemplo, si queremos recorrer un arreglo desde la posición 0 hasta la posición n-1, podemos utilizar la siguiente sintaxis:

```
for (int i = 0; i < n; i++) {
// Codigo a ejecutar
}
```

En esta sintaxis, “i” es el índice del arreglo y “n” es el tamaño del arreglo.

También es posible utilizar la notación “++i” en lugar de “i++” para incrementar el valor de la variable “i” en una unidad antes de que se ejecute el bloque de código. Por ejemplo:

```
for (int i = 0; i < n; ++i) {
// Codigo a ejecutar
}
```

Esta notación puede ser útil en algunos casos en los que se desea incrementar el valor de la variable antes de utilizarla en el bloque de código.

La `inicializacion` define una variable de control y su valor inicial. La `condicion` define la condición que debe cumplirse para continuar ejecutando el ciclo. El `incremento` se ejecuta después de cada iteración y permite modificar el valor de la variable de control. Si la condición es verdadera, se ejecutan las instrucciones dentro del bloque de código entre llaves. Después de cada iteración, se evalúa la condición nuevamente. Si la condición es falsa, se sale del ciclo.

1.5.4. Ciclo repetitivo: `while`

El ciclo repetitivo `while` permite ejecutar una o varias instrucciones mientras se cumpla una condición determinada. Su sintaxis es la siguiente:

```
while (condicion) {
// Instrucciones a ejecutar mientras se cumpla la condicion
}
```

Si la condición es verdadera, se ejecutan las instrucciones dentro del bloque de código entre llaves. Después de cada ejecución, se evalúa la condición nuevamente. Si la condición es falsa, se sale del ciclo.

1.5.5. Ciclo repetitivo: `do-while`

El ciclo repetitivo `do-while` es similar al ciclo `while`, pero la evaluación de la condición se realiza después de la primera ejecución del bloque de código. Su sintaxis es la siguiente:

```
do {
// bloque de codigo
} while (condicion);
```

En este caso, el bloque de código se ejecuta al menos una vez, independientemente de si la condición es verdadera o falsa en la primera evaluación. Después de la primera ejecución, la condición se evalúa nuevamente. Si es verdadera, el bloque de código se ejecuta de nuevo y así sucesivamente, hasta que la condición se vuelva falsa.

1.5.6. El comando `break`

En ocasiones, puede ser necesario detener un ciclo antes de que este termine su ejecución normal. Para esto, se utiliza el comando `break`.

El comando `break` permite salir de un ciclo de manera inmediata, incluso si la condición de finalización no se ha cumplido. Su sintaxis es la siguiente:

```
1  while (condicion) {
2    // cuerpo del ciclo
3    if (condicionDeSalida) {
4      break;
5    }
6  }
7
```

En este ejemplo, el ciclo `while` se ejecutará mientras la condición sea verdadera. Si se cumple la condición de salida, se ejecutará el comando `break`, que hará que el ciclo se detenga inmediatamente y se salga de él.

Es importante tener en cuenta que el comando `break` solo detiene el ciclo más interno en el que se encuentra. Si se están utilizando ciclos anidados, se puede utilizar el comando `break` para salir de uno de los ciclos, pero no de todos.

Es posible que en algunos casos sea necesario utilizar el comando `break` en conjunto con una estructura condicional, como se muestra en el siguiente ejemplo:

```
1  while (condicion) {
2    // cuerpo del ciclo
3    if (condicionDeSalida) {
4      if (otraCondicion) {
5        break;
6      } else {
7        // hacer algo
8      }
9    }
10 }
```

```

```

En este caso, si se cumple la condición de salida, se evalúa la otra condición. Si esta es verdadera, se ejecuta el comando `break`, lo que hace que se salga del ciclo `while`. Si la otra condición es falsa, se continúa con la ejecución normal del ciclo.

El uso adecuado del comando `break` puede ser de gran utilidad en situaciones en las que se necesite salir de un ciclo de manera inmediata. Sin embargo, es importante utilizarlo con cuidado, ya que su uso excesivo puede llevar a un código difícil de entender y mantener.

1.5.7. Ejemplos practicos

Te dejo un ejemplo de método de análisis numérico de bisección implementado en Java:

```java
public class BisectionMethod {

    public static double findRoot(double a, double b, double epsilon) {
        double mid = (a + b) / 2.0;
        while (Math.abs(a - b) > epsilon) {
            if (function(mid) == 0) {
                return mid;
            }
            if (function(a) * function(mid) < 0) {
                b = mid;
            } else {
                a = mid;
            }
            mid = (a + b) / 2.0;
        }
        return mid;
    }

    public static double function(double x) {
        // Define la funcion cuya raiz se quiere encontrar
        return Math.pow(x,2)-2;
    }

    public static void main(String[] args) {
        double a = 0;
        double b = 2;
        double epsilon = 0.001;
        double root = findRoot(a, b, epsilon);
```

```
        System.out.println("La raiz de la funcion es: " + root);
    }
}
```

Este programa calcula la raíz cuadrada de 2 usando el método de bisección. La función `f` define la función que se va a evaluar, en este caso $f(x) = x^2 - 2$. El método `bisection` toma como entrada los límites del intervalo, una tolerancia y el número máximo de iteraciones, y devuelve la aproximación de la raíz. El programa principal simplemente llama al método `bisection` con los parámetros deseados y muestra el resultado.

1.6. Funciones

En Java, una función se define utilizando la siguiente estructura:

```
[modificadorAcceso] tipoRetorno nombreFuncion (tipoParametro1 nombreParametro1, tipoParametro2
    nombreParametro2, ...) {
// Codigo de la funcion
return valorRetorno;
}
```

Donde:

- **modificadorAcceso** indica el nivel de acceso a la función, puede ser `public`, `private` o `protected`.
- **tipoRetorno** es el tipo de dato que devuelve la función, puede ser un tipo primitivo o una clase.
- **nombreFuncion** es el nombre de la función.
- **tipoParametro** es el tipo de dato del parámetro que recibe la función.
- **nombreParametro** es el nombre del parámetro que recibe la función.
- **valorRetorno** es el valor que devuelve la función, debe ser del tipo de dato indicado en `tipoRetorno`.

A continuación, se muestra un ejemplo de cómo se define una función en Java:

```
public int suma(int num1, int num2) {
int resultado = num1 + num2;
return resultado;
}
```

Esta función recibe dos parámetros de tipo `int` y devuelve un valor de tipo `int` que es la suma de los dos parámetros.

Es importante destacar que una función puede no tener parámetros y también puede no devolver ningún valor. En este caso, se utiliza el tipo de dato `void` para indicar que no se devuelve ningún valor. Por ejemplo:

```
public void imprimirHola() {
System.out.println("Hola!");
}
```

Esta función no recibe ningún parámetro y no devuelve ningún valor, simplemente imprime en pantalla el mensaje "Hola!".

1.6.1. Modificadores de acceso

Los modificadores de acceso en Java se utilizan para controlar el nivel de acceso a las clases, métodos y variables. Los cuatro modificadores de acceso que se pueden utilizar en Java son:

- `public`: Los miembros públicos se pueden acceder desde cualquier parte del código. Se pueden acceder desde la propia clase, desde cualquier otra clase en el mismo paquete, desde cualquier subclase y desde cualquier clase en otro paquete.
- `protected`: Los miembros protegidos se pueden acceder desde la propia clase, desde cualquier otra clase en el mismo paquete y desde cualquier subclase, pero no se pueden acceder desde una clase en otro paquete a menos que la clase en el otro paquete sea una subclase.
- `default`: Los miembros con acceso predeterminado (sin especificar ningún modificador) se pueden acceder desde la propia clase y desde cualquier otra clase en el mismo paquete, pero no se pueden acceder desde cualquier subclase o desde una clase en otro paquete.
- `private`: Los miembros privados solo se pueden acceder desde la propia clase. No se pueden acceder desde cualquier otra clase, ni siquiera de una subclase.

1.6.2. El alcance o Scope de variables

El alcance o scope de una variable se refiere a la parte del programa en la que se puede acceder a esa variable. Hay tres tipos de alcances de variables en Java:

- `Alcance de clase`: una variable de clase tiene un alcance que es toda la clase en la que se declara. Esto significa que la variable se puede acceder desde cualquier método en la clase y desde cualquier instancia de la clase.

- `Alcance de instancia`: una variable de instancia tiene un alcance que es la instancia de la clase en la que se declara. Esto significa que la variable se puede acceder desde cualquier método en la instancia de la clase, pero no desde otros objetos o instancias.

- `Alcance local`: una variable local tiene un alcance que es el bloque de código en el que se declara. Esto significa que la variable se puede acceder solo dentro del bloque de código en el que se declara, y no desde otros bloques de código o métodos.

Aquí hay algunos ejemplos de variables en diferentes alcances en Java:

```
public class ScopeExample {
// Alcance de clase
    static int variableDeClase = 1;
// Alcance de instancia
    int variableDeInstancia = 2;
    public void ejemploDeScope() {
// Alcance local
        int variableLocal = 3;
    System.out.println("Variable de clase: " + variableDeClase);
    System.out.println("Variable de instancia: " + variableDeInstancia);
    System.out.println("Variable local: " + variableLocal);
    }
}
```

En el ejemplo anterior, la variable `variableDeClase` tiene un alcance de clase, lo que significa que se puede acceder desde cualquier parte del programa que tenga acceso a la clase `ScopeExample`. La variable `variableDeInstancia` tiene un alcance de instancia, lo que significa que es accesible para todos los métodos dentro de la clase y puede ser utilizadas por cualquier objeto de esta clase. La variable `variableLoacal` tiene un alcance local, lo que significa que solo es accesiblet dentro de este método o bloque.

El alcance de una variable se puede limitar utilizando el modificador de acceso. Las variables públicas son accesibles desde cualquier parte del código, mientras que las variables protegidas solo son accesibles por la propia clase, las subclases y otras clases dentro del mismo paquete. Las variables privadas solo son accesibles por la propia clase y no pueden ser accedidas desde ninguna otra clase. Las variables sin modificador solo son accesibles dentro del mismo paquete.

1.7. Documentación con Javadoc

La documentación es una parte esencial de cualquier proyecto de programación, ya que permite a otros programadores entender el propósito, funcionamiento y uso del código. En Java, una herramienta muy

útil para generar documentación es Javadoc.

Javadoc es una herramienta que genera documentación en formato HTML a partir del código fuente y los comentarios del desarrollador. Los comentarios de Javadoc comienzan con el símbolo `/**` y terminan con el símbolo `*/`. El contenido de los comentarios de Javadoc se puede utilizar para generar documentación, incluyendo información sobre la clase, los campos, los métodos y los parámetros.

La documentación generada por Javadoc se puede ver en cualquier navegador web y se puede integrar en herramientas de desarrollo, como Eclipse o NetBeans, para permitir una fácil navegación a través del código y la documentación.

Para documentar una clase, un método o un campo con Javadoc, es necesario agregar comentarios Javadoc al código fuente. Los comentarios de Javadoc deben seguir ciertas convenciones y etiquetas para que la documentación se genere correctamente. Por ejemplo, para documentar un método, es necesario incluir la etiqueta `@param` para cada parámetro del método y la etiqueta `@return` para indicar el valor de retorno del método.

Un ejemplo de comentario Javadoc para documentar un método sería:

```
/**
Este metodo suma dos numeros enteros y devuelve el resultado.
*@param a El primer numero a sumar
*@param b El segundo numero a sumar
*@return La suma de a y b
*/

public int sumar(int a, int b) {
return a + b;
}
```

Javadoc es una herramienta útil para generar documentación a partir del código fuente y los comentarios del desarrollador. La documentación generada por Javadoc ayuda a otros programadores a entender el propósito, funcionamiento y uso del código, lo que es esencial para el mantenimiento y la colaboración en proyectos de programación.

1.7.1. Tags de Java Docs

Los tags de Javadoc se utilizan para proporcionar información adicional sobre una clase, un método o un campo, como el autor, la versión, los parámetros y el valor de retorno.

Los tags de Javadoc comienzan con el símbolo `@` y se colocan en el comentario Javadoc. Algunos de los tags de Javadoc más comunes incluyen:

- `@author`: especifica el autor de la clase o el método.
- `@version`: especifica la versión del código.
- `@param`: describe un parámetro de un método o constructor.
- `@return`: describe el valor de retorno de un método.
- `@throws`: describe las excepciones que se pueden lanzar desde un método.
- `@deprecated`: indica que un método o clase está obsoleto y se eliminará en una versión futura.
- `@see`: proporciona un enlace a otra clase, método o campo relacionado.

Por ejemplo, aquí hay un método documentado con algunos tags de Javadoc:

```
1  /**
2
3  Este metodo multiplica dos numeros enteros y devuelve el resultado.
4  @param a El primer numero a multiplicar
5  @param b El segundo numero a multiplicar
6  @return El producto de a y b
7  @throws IllegalArgumentException si a o b es negativo
8  @since 1.0
9  @deprecated Desde la versiun 2.0, usar el mutodo multiplicar(int a, int b, int c)
10 */
11 @Deprecated
12 public int multiplicar(int a, int b) {
13 if (a < 0 || b < 0) {
14 throw new IllegalArgumentException("a y b deben ser positivos");
15 }
16 return a * b;
17 }
18
```

En este ejemplo, el método está documentado con los tags `@param`, `@return` y `@throws`. También se utiliza el tag `@since` para indicar la versión en la que se introdujo el método y el tag `@deprecated` para indicar que el método se eliminará en una versión futura.

2. Programación orentada a objetos

La programación orientada a objetos ("POO") es un paradigma de programación que se enfoca en la creación de objetos que contienen tanto datos como funciones que actúan sobre esos datos. Esta técnica de programación se centra en la creación de *clases* que definen los objetos, y los objetos son instancias de esas clases.

En POO, los objetos son considerados como "cosas" que tienen ciertas propiedades y comportamientos. Cada objeto puede tener un conjunto de datos asociados (denominados "atributos" o "propiedades"), así como una serie de funciones o "métodos" que pueden realizar acciones sobre esos datos.

Una de las principales ventajas de la POO es que permite la modularidad del código, lo que significa que puedes dividir el código en pequeñas secciones que se pueden desarrollar, probar y mantener de forma independiente. Además, la POO también hace que el código sea más fácil de entender y mantener a largo plazo, ya que las funciones y datos relacionados están agrupados en objetos que tienen un propósito claro.

2.1. Encapsulación, Herencia, Polimorfismo y Abstracción

En POO, hay cuatro conceptos fundamentales que se deben entender:

1. *Encapsulación*: este concepto se refiere a la ocultación de datos y métodos dentro de un objeto para que solo puedan ser accedidos por el objeto mismo o por otros objetos que tengan una relación específica con él.

2. *Herencia*: la herencia permite crear una clase nueva a partir de una ya existente, pero añadiéndole o modificando sus propiedades y métodos.

3. *Polimorfismo*: este concepto se refiere a la capacidad de diferentes objetos de una misma clase para responder de manera distinta a un mismo mensaje o función.

4. *Abstracción*: la abstracción se refiere a la capacidad de crear una clase que representa un concepto o idea abstracta, sin necesidad de especificar todos los detalles en su definición.

La POO se utiliza en muchos lenguajes de programación populares, como Java, Python, C++ y muchos otros

2.1.1. Modelado

El modelado es una parte fundamental de la programación orientada a objetos. El objetivo del modelado es crear un modelo conceptual de un sistema o aplicación en términos de clases, objetos, atributos,

métodos y relaciones entre ellos.

En el modelado, se utilizan diagramas de clases para representar la estructura de un sistema. Un diagrama de clases es una representación gráfica de las clases en un sistema y las relaciones entre ellas. Las clases se representan como cajas con el nombre de la clase, sus atributos y métodos. Las relaciones entre las clases se representan mediante líneas que indican la naturaleza de la relación.

Además de los diagramas de clases, también se utilizan otros diagramas para modelar otros aspectos de la programación orientada a objetos. Por ejemplo, el diagrama de secuencia se utiliza para modelar la interacción entre objetos en un sistema, el diagrama de estado se utiliza para modelar el comportamiento de un objeto en diferentes estados y el diagrama de actividades se utiliza para modelar el flujo de trabajo de un proceso.

El modelado es una parte importante del desarrollo de software orientado a objetos porque ayuda a los desarrolladores a comprender mejor el sistema que están construyendo y a identificar y resolver problemas potenciales antes de escribir cualquier código. También ayuda a garantizar que el sistema sea modular, fácil de entender y mantener a largo plazo.

2.2. UML - Lenguaje de Modelado Unificado

UML ("Lenguaje de Modelado Unificado") es un lenguaje de modelado visual utilizado para el diseño y la documentación de sistemas de software orientados a objetos. UML es una herramienta popular utilizada por los desarrolladores de software para visualizar, especificar, construir y documentar los sistemas de software.

El lenguaje UML consta de varios tipos de diagramas, cada uno de los cuales se utiliza para modelar diferentes aspectos de un sistema de software. Estos diagramas incluyen:

1. Diagrama de Clases: representa la estructura estática del sistema, incluyendo las clases, los atributos y los métodos, y las relaciones entre ellos.
2. Diagrama de Objetos: representa una instancia específica de una clase en un punto específico del tiempo, incluyendo los valores de los atributos.
3. Diagrama de Casos de Uso: representa los actores, los casos de uso y las relaciones entre ellos.
4. Diagrama de Secuencia: representa la interacción entre los objetos en una secuencia de eventos.
5. Diagrama de Actividad: representa el flujo de trabajo de un proceso o algoritmo.
6. Diagrama de Estados: representa el comportamiento de un objeto en diferentes estados.

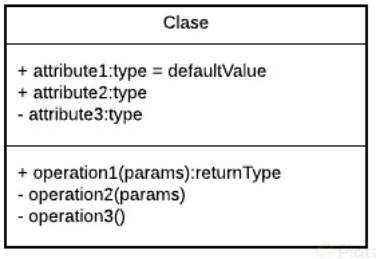
7. Diagrama de Componentes: representa los componentes del sistema y las relaciones entre ellos.

8. Diagrama de Despliegue: representa la distribución física del sistema en el hardware.

Estos diagramas son útiles para modelar diferentes aspectos de un sistema de software y proporcionan una visión completa del sistema desde diferentes perspectivas. Los diagramas de UML son útiles tanto para los desarrolladores como para los interesados en el sistema, como los gerentes de proyecto y los clientes, ya que proporcionan una visión clara y detallada del sistema.

2.2.1. Clases

Las clases se representan así:



En la parte superior se colocan los atributos o propiedades, y debajo las operaciones de la clase. Notarás que el primer caracter con el que empiezan es un símbolo. Este denotará la visibilidad del atributo o método, esto es un término que tiene que ver con Encapsulamiento y veremos más adelante a detalle.
Estos son los niveles de visibilidad que puedes tener:

1. − private

2. + public

3. # protected

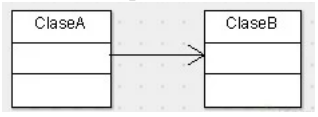
4. default

Una forma de representar las relaciones que tendrá un elemento con otro es a través de las flechas en UML, y aquí tenemos varios tipos, estos son los más comunes:

2.2.2. Asociación

Como su nombre lo dice, notarás que cada vez que esté referenciada este tipo de flecha significará que ese elemento contiene al otro en su definición.

La flecha apuntará hacia la dependencia.

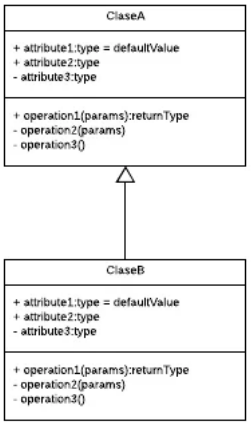


Con esto vemos que la ClaseA está asociada y depende de la ClaseB.

2.2.3. Herencia

Siempre que veamos este tipo de flecha se estará expresando la herencia.

La dirección de la flecha irá desde el hijo hasta el padre.

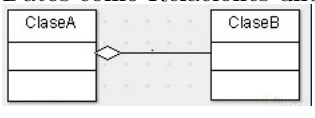


Con esto vemos que la ClaseB hereda de la ClaseA

2.2.4. Agregación

Este se parece a la asociación en que un elemento dependerá del otro, pero en este caso será: Un elemento dependerá de muchos otros.

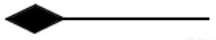
Aquí tomamos como referencia la multiplicidad del elemento. Lo que comúnmente conocerías en Bases de Datos como Relaciones uno a muchos.



Con esto decimos que la ClaseA contiene varios elementos de la ClaseB. Estos últimos son comúnmente representados con listas o colecciones de datos.

2.2.5. Composición

Este es similar al anterior solo que su relación es totalmente compenetrada.



De tal modo que conceptualmente una de estas clases no podría vivir si no existiera la otra.

3. JAVA orientada a objetos

Java es uno de los lenguajes de programación más populares y utilizados en el mundo, y también es conocido por su fuerte orientación a objetos. En Java, todo es un objeto, incluso los tipos primitivos, como los enteros y los caracteres, se representan como objetos.

La POO en Java se basa en la definición de clases y objetos. Una *clase* es un plano o plantilla que define la estructura, atributos y métodos de los objetos que se pueden crear a partir de ella. Un *objeto* es una instancia de una clase, es decir, un miembro concreto de la clase que contiene sus propios valores para los atributos definidos en la clase.

La encapsulación en Java se logra mediante el uso de modificadores de acceso, como `private`, `protected` y `public`, que permiten controlar el acceso a los atributos y métodos de una clase desde otras clases. La herencia se logra mediante la extensión de una clase existente mediante la palabra clave `extends`. El polimorfismo se logra mediante la implementación de interfaces o mediante la sobrecarga y la anulación de métodos.

La abstracción en Java se logra mediante el uso de interfaces y clases abstractas. Una *interface* es una colección de métodos sin implementación que una clase puede implementar para proporcionar su propia implementación. Una *clase abstracta* es una clase que no se puede instanciar directamente, sino que debe ser subclasificada y extendida.

3.1. sintaxis

La programación orientada a objetos en Java se basa en la definición de clases. Una clase se define con la palabra clave “class”, seguida del nombre de la clase y las llaves que encierran el cuerpo de la clase. Por ejemplo:

```
1 public class MiClase {
2     // cuerpo de la clase
3     }
```

Dentro de la clase, podemos definir atributos y métodos. Los atributos se definen con el tipo de dato y el nombre del atributo. Por ejemplo:

```
1 public class MiClase {
2     int miVariable;
3     }
```

En este ejemplo, se define un atributo “miVariable” de tipo entero.

Los métodos se definen con el tipo de retorno, el nombre del método, los parámetros que recibe entre paréntesis y el cuerpo del método encerrado entre llaves. Por ejemplo:

```
public class MiClase {
    int miVariable;
    public void miMetodo(int parametro) {
    // cuerpo del metodo
    }
}
```

En este ejemplo, se define un método "miMetodo" que no devuelve ningún valor ("void") y recibe un parámetro de tipo entero llamado "parametro".

Java también admite la herencia, que se define con la palabra clave "extends". Por ejemplo:

```
public class MiSubclase extends MiClase {
    // cuerpo de la subclase
}
```

En este ejemplo, se define una subclase llamada "MiSubclase" que extiende la clase "MiClase".

El polimorfismo también es una característica importante de la programación orientada a objetos en Java. Por ejemplo, podemos definir una variable de tipo de la clase base que puede referirse a una instancia de cualquiera de sus subclases. Por ejemplo:

```
MiClase miObjeto = new MiSubclase();
```

En este ejemplo, la variable "miObjeto" se declara como un objeto de tipo "MiClase", pero se inicializa como una instancia de la subclase "MiSubclase".

Finalmente, el encapsulamiento es una técnica importante en la programación orientada a objetos que permite ocultar la complejidad interna de una clase. En Java, podemos definir variables y métodos como privados con la palabra clave "private", lo que significa que solo se pueden acceder desde dentro de la clase. Por ejemplo:

```
public class MiClase {
    private int miVariable;
    private void miMetodoPrivado() {
        // cuerpo del metodo
    }
}
```

En este ejemplo, tanto la variable "miVariable" como el método "miMetodoPrivado" solo son accesibles desde dentro de la clase "MiClase".

3.1.1. Metodo constructor en una clase

Un constructor es un método especial que se utiliza para inicializar una instancia de una clase. El constructor se llama automáticamente cuando se crea una nueva instancia de la clase utilizando el operador "new". El constructor se define con el mismo nombre que la clase y no tiene un tipo de retorno explícito. Por ejemplo:

```
public class MiClase {
    int miVariable;
    public MiClase() {
        // cuerpo del constructor
    }

    public MiClase(int valor) {
        miVariable = valor;
    }
}
```

En este ejemplo, se define una clase llamada "MiClase" que contiene dos constructores: uno sin parámetros y otro que recibe un parámetro de tipo entero. El constructor sin parámetros simplemente tiene un cuerpo vacío, mientras que el constructor que recibe un parámetro inicializa la variable "miVariable" con el valor que se le pasa como argumento.

Es importante tener en cuenta que si se define un constructor para una clase, el compilador de Java no generará automáticamente un constructor predeterminado sin parámetros. Si se desea tener un constructor predeterminado, se debe definir explícitamente en la clase, como se muestra en el siguiente ejemplo:

```
public class MiClase {
    int miVariable;
    public MiClase() {
        // cuerpo del constructor sin parametros
    }

    public MiClase(int valor) {
        miVariable = valor;
    }

    // Constructor predeterminado
    public MiClase() {
        // cuerpo del constructor sin parametros
```

```
14      }
15  }
```

En este ejemplo, se define explícitamente un constructor predeterminado sin parámetros en la clase "MiClase", lo que permite crear instancias de la clase sin pasar ningún argumento al constructor.

3.2. Ejemplo práctico: Sistema de gestión de una biblioteca

Imaginemos que queremos crear un sistema de gestión de una biblioteca en Java. Para ello, podemos utilizar los conceptos de modularidad, abstracción, polimorfismo, herencia y encapsulamiento.

3.2.1. Modularidad

La modularidad se refiere a la división del código en módulos independientes que pueden ser desarrollados, probados y mantenidos por separado. En nuestro sistema de gestión de biblioteca, podemos utilizar la modularidad para dividir el código en diferentes clases, cada una de las cuales es responsable de una funcionalidad específica. Por ejemplo, podemos tener una clase para la gestión de libros, otra clase para la gestión de usuarios, y otra clase para la gestión de préstamos.

Abstracción

La abstracción se refiere a la capacidad de ocultar los detalles complejos y mostrar sólo la información esencial. En nuestro sistema de gestión de biblioteca, podemos utilizar la abstracción para crear una clase abstracta llamada `Item` que representa tanto a los libros como a otros elementos de la biblioteca, como DVDs o CDs. Esta clase podría contener métodos y propiedades que son comunes a todos los elementos de la biblioteca, como un número de identificación y un título.

Polimorfismo

El polimorfismo se refiere a la capacidad de los objetos de tomar diferentes formas. En nuestro sistema de gestión de biblioteca, podemos utilizar el polimorfismo para crear diferentes tipos de elementos de la biblioteca, como libros y DVDs, que heredan de la clase `Item` y tienen sus propios métodos y propiedades únicas. Podríamos tener una clase `Book` que hereda de la clase `Item` y tiene propiedades como el autor y la fecha de publicación, y una clase `DVD` que también hereda de `Item` y tiene propiedades como la duración y el género.

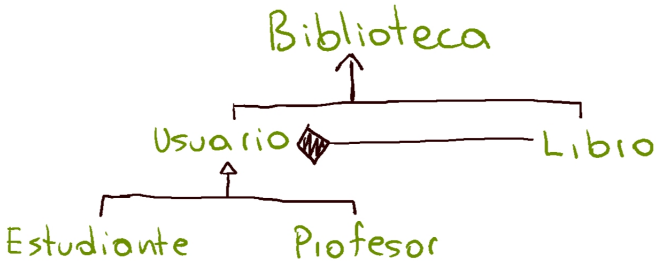
Herencia

La herencia se refiere a la capacidad de las clases hijas de heredar propiedades y métodos de la clase padre. En nuestro sistema de gestión de biblioteca, podemos utilizar la herencia para crear una jerarquía de clases en la que las clases hijas heredan propiedades y métodos de la clase padre. Por ejemplo, la clase `Book` podría heredar de la clase `Item` y tener acceso a sus propiedades y métodos.

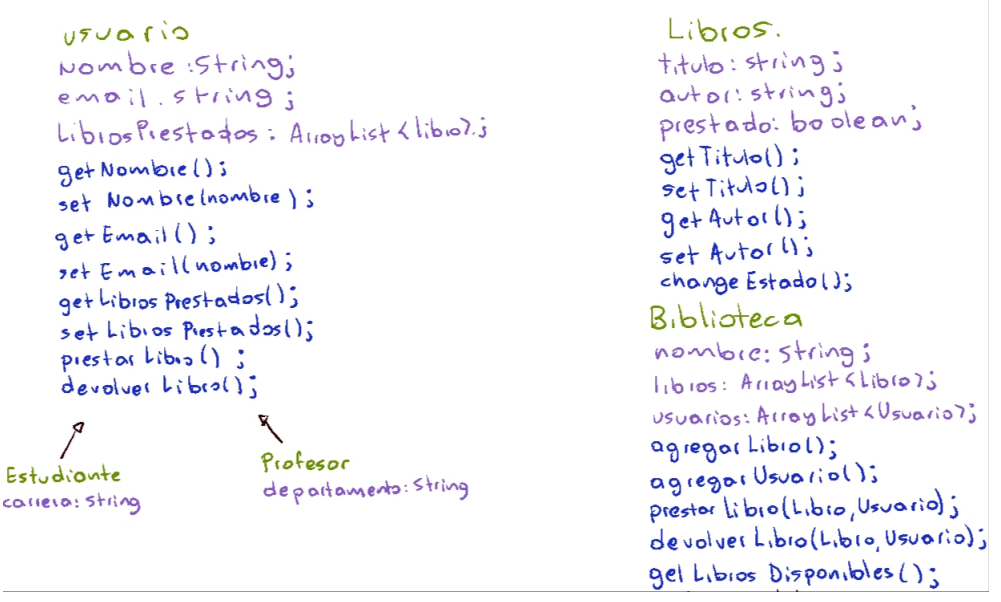
Encapsulamiento

El encapsulamiento se refiere a la capacidad de ocultar los detalles de implementación y proteger los datos y métodos de una clase de acceso no autorizado. En nuestro sistema de gestión de biblioteca, podemos utilizar el encapsulamiento para proteger los datos de los elementos de la biblioteca y limitar el acceso a ellos. Podemos definir las propiedades y métodos de la clase `Item` como privados para que sólo puedan ser accedidos y modificados por métodos dentro de la misma clase.

Para el ejemplo Es útil realizar el diagrama UML:



Donde



Comencemos a escribir codigo:

Creamos la clase Biblioteca:

```
1 import java.lang.reflect.Array;
2 import java.util.ArrayList;
3
4 public class Biblioteca {
5     private String nombre;
6     private ArrayList<Libro> libros;
7     private ArrayList<Usuario> usuarios;
8
9     public Biblioteca(String nombre) {
```

```
10          this.nombre = nombre;
11          libros = new ArrayList<Libro>();
12          usuarios = new ArrayList<Usuario>();
13      }
14
15      public void agregarLibro(Libro libro) {
16          libros.add(libro);
17      }
18
19      public void agregarUsuario(Usuario usuario) {
20          usuarios.add(usuario);
21      }
22
23      public void prestarLibro(Libro libro, Usuario usuario) {
24          if (libros.contains(libro) && !libro.prestado) {
25              libro.changeEstado();
26              usuario.prestarLibro(libro);
27              System.out.println("El libro " + libro.getTitulo() + " ha sido prestado a " + usuario.getNombre()
    );
28          } else {
29              System.out.println("No se puede prestar el libro " + libro.getTitulo() + " a " + usuario.
    getNombre());
30          }
31      }
32
33      public void devolverLibro(Libro libro, Usuario usuario) {
34          if (libros.contains(libro) && !libro.prestado) {
35              usuario.devolverLibro(libro);
36              usuario.devolverLibro(libro);
37              libro.changeEstado();
38              System.out.println("El libro " + libro.getTitulo() + " ha sido devuelto por " + usuario.getNombre
    ());
39          }
40      }
41      public ArrayList<Libro> getLibrosDisponibles() {
42          ArrayList<Libro> librosDisponibles = new ArrayList<Libro>();
43          for(Libro libro : libros){
44              if(!libro.prestado){
45                  librosDisponibles.add(libro);
46                  System.out.println("El libro " + libro.getTitulo() + " Esta disponible");
47              }
```

```
        }
        return librosDisponibles;
    }
}
```

Y hablando de herencia, podemos podemos agregar una clase Usuario y luego crear dos subclases Estudiante y Profesor que hereden de ella. De esta manera, los objetos de tipo Estudiante y Profesor tendrán todos los atributos y métodos de la clase Usuario, además de los propios.

```
import java.util.ArrayList;

class Usuario {
    private String nombre;
    private String email;
    private ArrayList<Libro>librosPrestados;
    public Usuario(String nombre, String email) {
        this.nombre = nombre;
        this.email = email;
        this.librosPrestados = new ArrayList<Libro>();
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public ArrayList<Libro> getLibrosPrestados() {
        return librosPrestados;
    }

```

```
33      public void setLibrosPrestados(ArrayList<Libro> librosPrestados) {
34          this.librosPrestados = librosPrestados;
35      }
36
37      public void prestarLibro(Libro libro) {
38          librosPrestados.add(libro);
39      }
40
41      public void devolverLibro(Libro libro) {
42          librosPrestados.remove(libro);
43      }
44
45      @Override
46      public String toString() {
47          return "Usuario [nombre=" + nombre + ", email=" + email + ", librosPrestados=" + librosPrestados + "]
        ";
48      }
49  }
50
51  class Estudiante extends Usuario {
52      private String carrera;
53
54      public Estudiante(String nombre, String email, String carrera) {
55          super(nombre,email);
56          this.carrera = carrera;
57      }
58
59      public String getCarrera() {
60          return carrera;
61      }
62  }
63
64  class Profesor extends Usuario {
65      private String departamento;
66
67      public Profesor(String nombre, String email, String carrera) {
68          super(nombre,email);
69          this.departamento = departamento;
70      }
71
72      public String getDepartamento() {
```

```
        return departamento;
    }
}
```

Aquí hemos creado dos subclases Estudiante y Profesor que heredan de la clase Usuario. Ahora podemos crear objetos de estas subclases y usar todos los métodos y atributos de la clase Usuario, además de los métodos y atributos adicionales de cada subclase. Por ejemplo, podemos crear un objeto Estudiante.

Pero también debemos crear la clase Libro:

```
class Libro {
    private String titulo;
    private String autor;
    public boolean prestado;

    public Libro(String titulo, String autor) {
        this.titulo = titulo;
        this.autor = autor;
        this.prestado = false;
    }
    public void changeEstado() {
        this.prestado = !prestado;
    }
    public String getTitulo() {
        return titulo;
    }

    @Override
    public String toString() {
        return "Libro [titulo=" + titulo + ", autor=" + autor +  ", prestado="
                + prestado + "]";
    }
}
```

Mostremos un ejemplo de cómo se podría instanciar y usar los métodos de las clases que hemos definido:

```
public class Main {
    public static void main(String[] args) {
        Biblioteca biblioteca = new Biblioteca("Biblioteca Municipal");

        // Agregamos algunos libros a la biblioteca
        Libro libro1 = new Libro("100 a˜nos de soledad", "Gabriel Garc'ia M'arquez");
```

```
        Libro libro2 = new Libro("El Aleph", "Jorge Luis Borges");
        Libro libro3 = new Libro("Rayuela", "Julio Cort'azar");
        biblioteca.agregarLibro(libro1);
        biblioteca.agregarLibro(libro2);
        biblioteca.agregarLibro(libro3);

        // Agregamos algunos usuarios a la biblioteca
        Usuario usuario1 = new Usuario("Juan", "P'erez");
        Usuario usuario2 = new Usuario("Mar'ia", "Gonz'alez");
        biblioteca.agregarUsuario(usuario1);
        biblioteca.agregarUsuario(usuario2);

        // Imprimimos la informaci'on del libro 1
        System.out.println(libro1);

        // Imprimimos los libros que est'an disponibles en la biblioteca
        System.out.println("Libros disponibles:");
        biblioteca.getLibrosDisponibles();
    }
```

Cuando lo ejecutemos la salida será:

```
Libro [titulo=100 años de soledad, autor=Gabriel García Márquez, prestado=false]
Libros disponibles:
El libro 100 años de soledad Está disponible
El libro El Aleph Está disponible
El libro Rayuela Está disponible


Process finished with exit code 0
```

4. Programación funcional

El paradigma de programación funcional es un enfoque de programación que se centra en el uso de funciones para resolver problemas y crear programas. A diferencia de los lenguajes de programación imperativos, que se centran en el cambio de estado y la ejecución de instrucciones secuenciales, los lenguajes de programación funcional se centran en la evaluación de expresiones y la aplicación de funciones.

En un lenguaje de programación funcional, las funciones son tratadas como valores de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como argumentos a otras funciones, y devueltas como valores de retorno. Las funciones en un lenguaje de programación funcional son definidas sin efectos secundarios, lo que significa que no modifican el estado de la aplicación ni tienen efectos impredecibles en otras partes del programa.

Uno de los beneficios del paradigma de programación funcional es que permite escribir código más conciso y fácil de entender, ya que se enfoca en la lógica de las funciones y no en los detalles de la implementación. Además, al no tener efectos secundarios, los programas escritos en lenguajes de programación funcional son más seguros y menos propensos a errores.

En el paradigma de programación funcional, podemos clasificar las funciones en varios tipos principales, que incluyen:

1. Funciones puras: Son funciones que no tienen efectos secundarios y siempre producen el mismo resultado cuando se les dan los mismos argumentos. No dependen del estado de la aplicación ni del entorno externo. Por lo tanto, las funciones puras no modifican ni leen el estado de la aplicación, y no interactúan con el mundo exterior. Estas funciones son muy importantes en la programación funcional porque se pueden utilizar en programas concurrentes y paralelos sin preocuparse por problemas de estado compartido.

2. Funciones de alto orden: Son funciones que toman otras funciones como argumentos y/o devuelven funciones como resultado. Estas funciones son muy útiles en la programación funcional porque nos permiten abstraer la lógica común en funciones genéricas y reutilizarlas para diferentes propósitos.

3. Funciones recursivas: Son funciones que se llaman a sí mismas durante su ejecución. Estas funciones se utilizan para expresar problemas que se pueden descomponer en subproblemas más pequeños. La recursión se detiene cuando se alcanza un caso base que se puede resolver sin más recursión.

4. Funciones lambda: Son funciones anónimas que se pueden definir sin nombre. Se utilizan en la programación funcional para expresar funciones pequeñas y simples de manera concisa y sin crear una nueva clase para cada función.

4.1. Funciones puras

Para introducir el concepto de **función pura** debemos entender que un **efecto secundario** es todo cambio observable desde fuera del sistema

Dentro del paradigma de programación funcional, las funciones puras son una parte fundamental. Una función pura es una función que siempre produce el mismo resultado cuando se le dan los mismos argumentos, y no tiene efectos secundarios en el estado de la aplicación ni en el entorno externo.

Esto significa que una función pura no cambia ninguna variable global ni realiza ninguna operación de entrada/salida, como escribir en un archivo o leer de un dispositivo de entrada. En cambio, una función pura solo toma sus argumentos, realiza sus cálculos y devuelve un resultado.

Un beneficio clave de las funciones puras es que son predecibles y fáciles de razonar. Debido a que no tienen efectos secundarios, se pueden utilizar en cualquier parte del código sin preocuparse por cambios inesperados en el estado de la aplicación. Además, las funciones puras son fáciles de probar, ya que siempre producirán el mismo resultado con los mismos argumentos.

En contraposición, las funciones impuras son aquellas que pueden tener efectos secundarios y pueden producir diferentes resultados cuando se les dan los mismos argumentos. Estas funciones pueden ser más difíciles de razonar y probar, y pueden ser propensas a errores.

Por ejemplo:

- $f(x) = x^3$
- La función suma es una función pura pues lo unico que hace es calcular la suma de el par (a, b), no hay efectos secundarios ni depende de otra cosa que no sea la entrada.

```
1 public static int sumar(int a, int b) {
2     return a + b;
3 }
```

- la función que compara dos números es una función pura.

```
1 public static boolean es_mayor_que(int a, int b) {
2     return a > b;
3 }
```

4.2. Funciones de alto orden

Las funciones de alto orden son aquellas funciones que pueden recibir otras funciones como argumentos y/o retornar funciones como resultado. En el paradigma de programación funcional, las funciones

de alto orden son fundamentales para la creación de abstracciones y la composición de programas.

En Java, se pueden definir funciones de alto orden utilizando interfaces funcionales. Una interfaz funcional es una interfaz que declara exactamente un método abstracto y se puede utilizar para definir funciones lambda o referencias a métodos.

A continuación, te muestro un ejemplo de una función de alto orden en Java que recibe una función como argumento:

Supongamos que queremos definir una función que tome un array de enteros y una función que defina cómo se debe transformar cada elemento del array. La función debe devolver un nuevo array con los elementos transformados. Para ello, podemos definir la siguiente interfaz funcional:

```
public interface Transformador<T> {
    T aplicar(T valor);
}
```

Esta interfaz define un único método abstracto aplicar que toma un valor de tipo T y devuelve un valor del mismo tipo T. Esta interfaz se puede utilizar para definir funciones lambda o referencias a métodos que transformen valores de cualquier tipo.

A continuación, definimos la función de alto orden que utiliza la interfaz Transformador:

```
public class FuncionesDeAltoOrden {
    public static <T> T[] transformar(T[] array, Transformador<T> transformador) {
        T[] resultado = Arrays.copyOf(array, array.length);
        for (int i = 0; i < array.length; i++) {
            resultado[i] = transformador.aplicar(array[i]);
        }
        return resultado;
    }

    public static void main(String[] args) {
        Integer[] numeros = {1, 2, 3, 4, 5};
        String[] strings = {"uno", "dos", "tres", "cuatro", "cinco"};

        Transformador<Integer> multiplicarPorDos = n -> n * 2;
        Transformador<String> convertirAMayusculas = s -> s.toUpperCase();

        Integer[] resultado1 = transformar(numeros, multiplicarPorDos);
        String[] resultado2 = transformar(strings, convertirAMayusculas);

        System.out.println(Arrays.toString(resultado1));
        System.out.println(Arrays.toString(resultado2));
```

```
    }
}
```

En este ejemplo, la función transformar recibe un array de cualquier tipo T y una función transformador que toma un valor de tipo T y devuelve un nuevo valor de tipo T. La función transformar utiliza un bucle for para llamar a la función transformador con cada elemento del array y guardar el resultado en un nuevo array que se devuelve al final.

En el método main, se definen dos arrays, uno de enteros y otro de cadenas. Luego se definen dos funciones multiplicarPorDos y convertirAMayusculas que implementan la interfaz Transformador para multiplicar por dos cada número y convertir cada cadena a mayúsculas, respectivamente.

Finalmente, se llaman a la función transformar con cada array y la función correspondiente, y se imprimen los resultados.

Este ejemplo muestra cómo se pueden utilizar funciones de alto orden y una interfaz funcional propia para crear una función genérica y reutilizable que transforme arrays de cualquier tipo. Además, se demuestra la flexibilidad y potencia que se obtiene al utilizar este paradigma de programación.

4.3. Funciones Recursivas

Las funciones recursivas en Java son aquellas funciones que se llaman a sí mismas durante su ejecución. Es decir, en lugar de realizar un bucle para repetir una operación, la función se llama a sí misma con una entrada diferente hasta alcanzar un caso base donde la recursión termina.

Un ejemplo común de una función recursiva en Java es la implementación de la función factorial, que calcula el producto de todos los enteros positivos desde 1 hasta un número dado n. La implementación recursiva de la función factorial en Java es la siguiente:

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

En esta implementación, la función factorial se llama a sí misma con un argumento menor en cada llamada hasta que se alcanza el caso base donde n es igual a 0. El caso base devuelve el valor 1, lo que detiene la recursión. Luego, cada llamada recursiva multiplica n por el resultado de la llamada recursiva con n-1.

Una función recursiva que no sea pura puede ser aquella que modifica una variable global o que

interactúa con el entorno externo. A continuación, te muestro un ejemplo de una función recursiva en Java que modifica una variable global:

```
public class Recursivas {
    private static int contador = 0;
    public static int contarElementos(List<Integer> lista) {
        if (lista.isEmpty()) {
            return contador;
        } else {
            contador++;
            lista.remove(0);
            return contarElementos(lista);
        }
    }
}
```

En este ejemplo, la función `contarElementos` recibe una lista de enteros como argumento y utiliza recursión para contar cuántos elementos tiene la lista. Sin embargo, también modifica la variable global contador para llevar la cuenta.

Esta función no es pura porque su comportamiento depende del valor de la variable global contador y puede cambiar su valor cada vez que se llama a la función.

Es importante tener cuidado al usar funciones recursivas y asegurarse de que se alcance el caso base eventualmente para evitar una recursión infinita.

4.4. Funciones lambda

Las lambdas son una característica introducida en Java 8 que permite escribir código de manera más concisa y legible, en particular para operaciones con colecciones de objetos. En lugar de crear una clase anónima para definir una función, se puede utilizar una lambda para pasar una expresión de función directamente como argumento a un método. La sintaxis de una lambda es la siguiente:

```
(parameter1, parameter2, ...) -> { statement1; statement2; ... }
```

Donde `parameter1, parameter2, ...` son los parámetros de la función y `statement1, statement2, ...` son las instrucciones que se ejecutan en el cuerpo de la función.

Por ejemplo, la siguiente lambda se utiliza para imprimir todos los elementos de una lista:

```
(int a, int b) -> { return a + b; }
```

También se pueden utilizar métodos de referencia para invocar un método existente como una lambda. Por ejemplo, para imprimir todos los elementos de una lista con un método de referencia, se puede

hacer lo siguiente:

```
names.forEach(System.out::println);
```

En general, las lambdas permiten escribir código más expresivo y legible, y son especialmente útiles en el contexto de las colecciones de objetos.

Las lambdas en Java se pueden utilizar en diferentes contextos, como por ejemplo:

```
// Ejemplo 1: Ordenar una lista de strings usando lambdas
List<String> lista = Arrays.asList("Juan", "Pedro", "Mara");
Collections.sort(lista, (s1, s2) -> s1.compareTo(s2));
System.out.println(lista); // Salida: [Juan, Mara, Pedro]

// Ejemplo 2: Filtrar elementos de una lista usando lambdas
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> numerosPares = numeros.stream()
.filter(n -> n % 2 == 0)
.collect(Collectors.toList());
System.out.println(numerosPares); // Salida: [2, 4, 6, 8]

// Ejemplo 3: Crear un thread usando lambdas
Thread hilo = new Thread(() -> {
System.out.println("Este es un mensaje desde un hilo");
});
hilo.start();
```

Este código muestra tres ejemplos diferentes de cómo usar lambdas en Java: ordenar una lista de strings, filtrar elementos de una lista y crear un thread. En cada ejemplo se utiliza una sintaxis diferente para definir la lambda, pero en todos los casos se logra escribir código más conciso y legible.

Las funciones lambda son útiles porque permiten definir funciones de manera concisa y expresiva en el lugar donde se necesitan, sin tener que crear una clase separada para cada función. Esto hace que el código sea más fácil de leer, escribir y mantener.

Además, las funciones lambda son especialmente útiles cuando se utilizan junto con funciones de alto orden, ya que permiten pasar funciones como argumentos a otras funciones y devolver funciones como resultado de otras funciones. Esto facilita la implementación de patrones comunes de programación funcional, como el mapeo, filtrado, reducción y composición de funciones.

Otra ventaja de las funciones lambda es que pueden mejorar el rendimiento del programa al reducir la cantidad de código que se necesita para crear objetos temporales y reducir la sobrecarga de la recolección de basura. Las funciones lambda también pueden ser evaluadas de manera diferida, lo que significa que no

se evalúan hasta que se necesitan, lo que puede mejorar la eficiencia en algunos casos.

4.5. Datos mutables e inmutables

En programación, los datos mutables e inmutables se refieren a la capacidad de los datos para cambiar después de su creación.

Los datos mutables son aquellos que pueden ser modificados después de su creación. Por ejemplo, los arreglos en Java son mutables, lo que significa que se pueden modificar sus elementos después de haber sido creados. Los objetos también pueden ser mutables, lo que significa que se pueden modificar sus propiedades o atributos después de haber sido creados.

Por otro lado, los datos inmutables son aquellos que no pueden ser modificados después de su creación. Por ejemplo, las cadenas de caracteres en Java son inmutables, lo que significa que no se pueden modificar después de haber sido creadas. Los números también son inmutables en Java.

Los datos inmutables son más seguros en términos de concurrencia y paralelismo, ya que no pueden ser modificados por otros hilos mientras se están utilizando en un proceso. Además, los datos inmutables pueden ser compartidos entre varios hilos sin preocuparse por conflictos de modificación. Por otro lado, los datos mutables pueden causar problemas en el código si no se manejan adecuadamente, ya que pueden ser modificados en cualquier lugar del programa y hacer que el resultado final sea impredecible.

En general, es recomendable utilizar datos inmutables siempre que sea posible para evitar problemas de concurrencia y paralelismo y hacer que el código sea más seguro y predecible. Sin embargo, en algunos casos, los datos mutables pueden ser necesarios para realizar cambios en el estado de la aplicación o para mejorar el rendimiento. Es importante tener en cuenta los pros y los contras de cada enfoque y elegir el más adecuado para cada situación.

Un ejemplo de un dato mutable en Java es un arreglo. Supongamos que tenemos el siguiente arreglo:

```
int[] numeros = {1, 2, 3, 4};
```

Podemos modificar los elementos del arreglo en cualquier momento utilizando el índice correspondiente:

```
numeros[0] = 5;
```

Esto cambia el primer elemento del arreglo de 1 a 5.

Por otro lado, haciendo:

```
final int edad = 18;
```

Al utilizar la palabra clave final, estamos indicando que la variable no se puede modificar después de su creación. Por lo tanto, edad es un dato inmutable.

5. Java avanzado

5.1. Clases abstractas e interfaces

Las clases abstractas y las interfaces son dos herramientas importantes en la programación orientada a objetos de Java. Ambas permiten definir una estructura común para un conjunto de clases relacionadas, pero difieren en la forma en que se definen y se utilizan.

Las clases abstractas se utilizan como plantillas para otras clases que sí se pueden instanciar. Estas clases contienen métodos abstractos (es decir, métodos sin implementación) y métodos concretos (con implementación). Las clases hijas deben implementar todos los métodos abstractos definidos en la clase padre. De esta manera, se garantiza que la clase hija tenga una funcionalidad coherente con la clase padre.

Por otro lado, las interfaces son un tipo especial de clase abstracta que define sólo métodos abstractos. Estas interfaces se utilizan para definir un conjunto de métodos que deben ser implementados por cualquier clase que implemente la interfaz. De esta manera, se asegura que todas las clases que implementan la interfaz tengan un comportamiento consistente y coherente en todo momento.

Tanto las clases abstractas como las interfaces permiten establecer una jerarquía de clases y definir una estructura común para un conjunto de clases relacionadas. Sin embargo, la elección entre utilizar una clase abstracta o una interfaz depende de la situación específica y de las necesidades del proyecto en cuestión.

En esta sección, profundizaremos en la sintaxis y uso de las interfaces.

5.1.1. Clases abstractas

Las clases abstractas en Java son clases que no se pueden instanciar directamente, sino que se utilizan como plantillas para otras clases que sí se pueden instanciar. Es decir, son clases que sirven como base para otras clases más específicas.

Una clase abstracta se define utilizando la palabra clave `abstract`, y puede contener métodos abstractos (es decir, métodos sin implementación) y métodos concretos (con implementación). Los métodos abstractos deben ser implementados por las clases hijas, mientras que los métodos concretos pueden ser heredados o sobrescritos.

Una clase que extiende de una clase abstracta debe implementar todos los métodos abstractos definidos en la clase padre. De esta manera, se garantiza que la clase hija tenga una funcionalidad coherente con la clase padre.

Veamos un ejemplo sencillo:

```
1 public abstract class Figura {
```

```
    protected String color;
    public Figura(String color) {
    this.color = color;
    }

    public abstract double area();
    public abstract double perimetro();
}
```

En este ejemplo, la clase `Figura` es abstracta y define dos métodos abstractos (`area()` y `perimetro()`). También define un constructor que recibe un parámetro `color`. Esta clase es una plantilla para otras clases que representen figuras geométricas concretas (como círculos, rectángulos, etc.).

A continuación, se muestra un ejemplo de una clase hija que extiende de la clase `Figura`:

```
    public class Circulo extends Figura {
    private double radio;
    public Circulo(String color, double radio) {
    super(color);
    this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * Math.pow(radio, 2);
    }

    @Override
    public double perimetro() {
        return 2 * Math.PI * radio;
    }
}
```

En este ejemplo, la clase `Circulo` extiende de la clase `Figura` y define los métodos abstractos `area()` y `perimetro()`. Además, define un constructor que recibe dos parámetros (`color` y `radio`), y una variable de instancia `radio`.

Como puedes ver, las clases abstractas son una herramienta útil para definir una estructura común para un conjunto de clases relacionadas, y asegurarse de que todas las clases hijas tengan una funcionalidad coherente y consistente.

5.1.2. Interfaces

Una interfaz es una colección de métodos abstractos y constantes que se utilizan para definir un conjunto de comportamientos que deben ser implementados por una clase.

La sintaxis básica para declarar una interfaz es la siguiente:

```
public interface MiInterfaz {
    public void metodo1();
    public int metodo2();
}
```

En este ejemplo, se define una interfaz llamada "MiInterfaz" que tiene dos métodos abstractos: "metodo1" y "metodo2". Estos métodos no tienen una implementación definida y las clases que implementen esta interfaz deben proporcionar una implementación para ellos.

Las interfaces también pueden incluir constantes, que se definen utilizando la palabra clave "final":

```
public interface OtraInterfaz {
    public static final int CONSTANTE = 42;
    public void metodo3();
}
```

En este ejemplo, se define una interfaz llamada "OtraInterfaz" que tiene una constante llamada "CONSTANTE" y un método abstracto llamado "metodo3".

Para implementar una interfaz en una clase, se utiliza la palabra clave "implements". Por ejemplo:

```
public class MiClase implements MiInterfaz {
    public void metodo1() {
    System.out.println("Implementando metodo1...");
    }
    public int metodo2() {
    return 42;
    }
}
```

En este ejemplo, se define una clase llamada "MiClase" que implementa la interfaz "MiInterfaz". La clase proporciona una implementación para los métodos abstractos de la interfaz.

Las interfaces son una herramienta poderosa en Java que se utilizan para definir comportamientos comunes que deben ser implementados por diferentes clases. Esto permite crear código más modular y reutilizable en una amplia variedad de aplicaciones.

5.2. Manejo de excepciones

El manejo de excepciones es una técnica utilizada en Java para manejar errores y excepciones que puedan ocurrir durante la ejecución de un programa. Las excepciones son eventos que indican que algo inesperado ha ocurrido en el programa y que necesita ser manejado de alguna manera.

En Java, se utiliza el bloque "try-catch-finally"para manejar excepciones. El bloque “try” contiene el código que puede generar una excepción, mientras que el bloque “catch” maneja la excepción si se produce. El bloque “finally” se utiliza para ejecutar código que debe ejecutarse independientemente de si se produjo una excepción o no.

El siguiente es un ejemplo de cómo manejar una excepción en Java:

```
try {
// codigo que puede generar una excepcion
} catch (Excepcion1 e1) {
// manejo de la excepcion Excepcion1
} catch (Excepcion2 e2) {
// manejo de la excepcion Excepcion2
} finally {
// codigo que se ejecutara independientemente de si se produce una excepcion o no
}
```

En este ejemplo, el bloque “try” contiene el código que puede generar una excepción. Si se produce la excepción .ExcepcionExcepcion1", se manejará en el primer bloque, si se produce la excepción .ExcepcionExcepcion2", se manejará en el segundo bloque. Finalmente se ejecitara el terder bloque.

5.2.1. Tipo de excepciones

En Java, existen dos tipos de excepciones:

- Excepciones comprobadas (*checked exceptions*): son aquellas que el compilador obliga a manejar. Estas excepciones suelen ser producidas por acciones que pueden fallar debido a factores externos como la entrada/salida de datos, conexiones a bases de datos, entre otros. Para manejar estas excepciones, se debe utilizar la palabra clave `try-catch` o propagar la excepción con la palabra clave `throws`.
- Excepciones no comprobadas (*unchecked exceptions*): son aquellas que no están obligadas a ser manejadas por el compilador. Estas excepciones suelen ser producidas por errores en tiempo de ejecución, como la división entre cero (`ArithmeticException`), acceso a un índice inválido en un arreglo (`ArrayIndexOutOfBoundsException`), entre otros. Aunque no es obligatorio manejar estas excepciones, es recomendable hacerlo para evitar que el programa se detenga inesperadamente.

Es importante tener en cuenta que el manejo adecuado de excepciones es una parte esencial de la programación en Java, ya que permite detectar y corregir errores en el programa de manera más efectiva.

5.3. Expresiones regulares

Las expresiones regulares son patrones de búsqueda de texto que se utilizan para encontrar una secuencia específica de caracteres en un string. Java tiene una clase llamada `Pattern` y una clase llamada `Matcher` que se utilizan para trabajar con expresiones regulares.

5.3.1. Sintaxis de las expresiones regulares

La sintaxis de las expresiones regulares en Java es similar a la de otros lenguajes de programación y herramientas como Perl, Python y grep. A continuación se presentan algunos de los elementos básicos de la sintaxis de las expresiones regulares en Java:

- Los caracteres literales se pueden incluir en una expresión regular para buscar una coincidencia exacta. Por ejemplo, la expresión regular `abc` buscará la cadena "abc".
- Los caracteres especiales se utilizan para buscar patrones más complejos. Algunos de los caracteres especiales más comunes incluyen el punto (.) que coincide con cualquier carácter, el asterisco (*) que coincide con cero o más repeticiones del carácter anterior, y el signo de interrogación (?) que coincide con cero o una repetición del carácter anterior.
- Los corchetes se utilizan para buscar un conjunto de caracteres. Por ejemplo, la expresión regular `[aeiou]` buscará cualquier vocal.
- Las llaves se utilizan para buscar repeticiones específicas de un carácter o un conjunto de caracteres. Por ejemplo, la expresión regular `a3` buscará tres repeticiones de la letra "a".

5.3.2. Uso de expresiones regulares en Java

Java proporciona la clase `Pattern` para compilar expresiones regulares en objetos que se pueden reutilizar para buscar en múltiples cadenas. La clase `Matcher` se utiliza para buscar coincidencias en cadenas específicas.

A continuación se muestra un ejemplo simple de uso de expresiones regulares en Java:

```
1 import java.util.regex.*;
2
3 public class RegularExpressionsExample {
```

```
public static void main(String[] args) {
String input = "La respuesta es 42.";
String pattern = ".42.";
boolean isMatch = Pattern.matches(pattern, input);
System.out.println(isMatch); // true
}
}
```

Este ejemplo compila la expresión regular ‘‘.42.’’ y la utiliza para buscar la cadena “La respuesta es 42.” La función `Pattern.matches()` devuelve un valor booleano que indica si la cadena coincide con la expresión regular.

Las expresiones regulares son secuencias de caracteres que se utilizan para buscar y manipular patrones de texto. En Java, las expresiones regulares se implementan mediante la clase Pattern y Matcher del paquete java.util.regex.

El siguiente ejemplo muestra cómo usar expresiones regulares en Java para buscar una cadena que comienza con la letra "H"seguida de cualquier carácter y luego la letra .º":

```
String input = "Hola mundo";
Pattern pattern = Pattern.compile("H.o");
Matcher matcher = pattern.matcher(input);
if (matcher.find()) {
    System.out.println("Se encontro la cadena " + matcher.group());
} else {
    System.out.println("No se encontro la cadena");
}
```

En este ejemplo, la expresión regular "H.o"se compila en un patrón y se utiliza para crear un objeto Matcher. El método find() busca la siguiente subsecuencia del texto de entrada que coincide con el patrón y devuelve verdadero si se encuentra una coincidencia. El método group() devuelve la subcadena que coincide con el patrón.

Las expresiones regulares también se pueden utilizar para validar entradas de usuario, como direcciones de correo electrónico o números de teléfono. Por ejemplo, la siguiente expresión regular valida una dirección de correo electrónico:

```
String input = "user@example.com";
Pattern pattern = Pattern.compile("\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z]{2,}\b");
Matcher matcher = pattern.matcher(input);
if (matcher.find()) {
    System.out.println("La direccion de correo electronico es valida");
```

```
} else {
    System.out.println("La direccion de correo electronico no es valida");
    }
```

En este ejemplo, la expresión regular valida la sintaxis de una dirección de correo electrónico, verificando que contenga un nombre de usuario, un signo "@", un nombre de dominio y una extensión de dominio válida.

Las expresiones regulares son una herramienta poderosa para el procesamiento de texto en Java. Con un conocimiento sólido de su sintaxis y uso, puedes manejar de manera efectiva patrones de texto complejos.

5.4. API (Application Programming Interface)

- Una API (Application Programming Interface) es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con una aplicación o servicio externo mediante un conjunto de reglas y protocolos.
- Las APIs pueden tener diferentes tipos de solicitudes y respuestas, como GET, POST, PUT y DELETE. También pueden devolver datos en diferentes formatos, como JSON o XML.
- Para crear una API en Java, puedes utilizar una biblioteca como Spring Boot, que te permite crear una aplicación web y definir controladores que manejen las solicitudes entrantes y generen respuestas adecuadas.
- Los controladores pueden recibir parámetros en la URL, en la solicitud HTTP o en el cuerpo de la solicitud, y devolver diferentes tipos de respuestas, como objetos JSON o vistas HTML.

5.4.1. Tipos de API

Existen diferentes tipos de API que se utilizan en diferentes contextos, aquí te mencionamos algunos de los más comunes:

- **API REST:** Representational State Transfer (REST) es un tipo de arquitectura de software que utiliza HTTP para enviar y recibir datos. Las API REST permiten que diferentes sistemas se comuniquen entre sí a través de solicitudes HTTP, utilizando verbos HTTP como GET, POST, PUT y DELETE para enviar y recibir datos en diferentes formatos como JSON o XML.

- **API SOAP:** Simple Object Access Protocol (SOAP) es un protocolo basado en XML utilizado para intercambiar datos entre diferentes sistemas. Las API SOAP utilizan mensajes XML para comunicarse, y se basan en un conjunto de reglas y protocolos para enviar y recibir datos.

- **API GraphQL:** GraphQL es un lenguaje de consulta de datos utilizado para comunicar diferentes sistemas. Las API GraphQL permiten que los clientes definan las consultas que desean hacer y recibir solo los datos solicitados, lo que puede ser muy eficiente en términos de ancho de banda y tiempo de respuesta.

5.4.2. Protocolos utilizados en API

Las API utilizan diferentes protocolos para definir cómo se accede y manipula la información. Algunos de los protocolos más comunes utilizados en las API son:

- **HTTP (Hypertext Transfer Protocol)**: Este protocolo es utilizado por las API de REST para transferir datos entre el cliente y el servidor. Las solicitudes HTTP pueden ser GET, POST, PUT, DELETE, entre otras.

- **HTTPS (Hypertext Transfer Protocol Secure)**: Este protocolo es una versión segura del HTTP que utiliza SSL o TLS para cifrar la información transferida. Las API que utilizan HTTPS son más seguras y protegen la información del usuario.

- **XML (Extensible Markup Language)**: Este formato de mensaje es utilizado por las API SOAP para definir la estructura y el contenido de las solicitudes y respuestas. XML es un formato muy flexible y puede ser utilizado para una gran variedad de tipos de datos.

- **JSON (JavaScript Object Notation)**: Este formato de mensaje es utilizado por las API de REST para definir la estructura y el contenido de las solicitudes y respuestas. JSON es un formato muy popular y fácil de usar en aplicaciones web.

5.4.3. Utilidades de las API

Las API tienen diversas utilidades y beneficios tanto para los desarrolladores como para los usuarios. Algunas de las principales utilidades son:

- **Integración con otros sistemas**: Las API permiten la integración de diferentes sistemas y servicios. Esto es muy útil para los desarrolladores, ya que les permite crear aplicaciones que se comuniquen con servicios o aplicaciones externas y accedan a datos y funcionalidades de dichos servicios o aplicaciones.

Por ejemplo, una aplicación de transporte que utiliza la API de Google Maps para mostrar las rutas y direcciones a los usuarios.

- **Acceso a datos y funcionalidades específicas**: Las API permiten el acceso a datos y funcionalidades específicas de una aplicación o servicio externo. Esto permite a los desarrolladores crear aplicaciones que utilicen los datos y funcionalidades de diferentes fuentes. Por ejemplo, una aplicación de finanzas personales que utiliza la API de una institución financiera para obtener los saldos de las cuentas de los usuarios.

- **Desarrollo de aplicaciones de manera más rápida**: Las API permiten a los desarrolladores acelerar el proceso de desarrollo al no tener que construir desde cero todas las funcionalidades y características que necesitan para sus aplicaciones. Esto les permite concentrarse en las características únicas de sus aplicaciones, en lugar de pasar tiempo en la creación de funcionalidades que ya están disponibles en otras aplicaciones o servicios.

- **Mejora de la experiencia del usuario**: Las API pueden ayudar a mejorar la experiencia del usuario en una aplicación o servicio, ya que permiten acceder a datos y funcionalidades adicionales de una manera más rápida y fácil. Por ejemplo, una aplicación de compras que utiliza la API de un servicio de pagos para permitir a los usuarios pagar con su método de pago preferido.

- **Mejora de la calidad de los datos**: Las API pueden ayudar a mejorar la calidad de los datos de una aplicación o servicio, ya que permiten acceder a fuentes de datos actualizadas y precisas. Por ejemplo, una aplicación de clima que utiliza la API de una fuente de datos confiable para obtener las condiciones climáticas actuales.

- **Aumento de la seguridad**: Las API pueden ayudar a aumentar la seguridad de una aplicación o servicio, ya que permiten implementar autenticación y autorización para restringir el acceso a los datos y funcionalidades sólo a los usuarios autorizados.

- **Mejora de la escalabilidad**: Las API pueden ayudar a mejorar la escalabilidad de una aplicación o servicio, ya que permiten que múltiples aplicaciones y servicios accedan a los datos y funcionalidades de una fuente centralizada de manera simultánea, sin causar una sobrecarga en el sistema.

Las API son una herramienta esencial para el desarrollo de aplicaciones modernas y permiten la integración y el acceso a datos y funcionalidades de diferentes servicios y aplicaciones externas, lo que resulta en un mejor rendimiento, seguridad, escalabilidad y experiencia del usuario.

5.5. Ejemplo practico

Supongamos que queremos obtener la información de una API pública que devuelve la cotización del dólar en tiempo real.

```java
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) {
        try {
            String apiKey = "NZ33T3sO6yCVPtxJmVyYZmY4iPGj1mJX";
            String url = "https://api.apilayer.com/exchangerates_data/convert?to=EUR&from=USD&amount=100";
            URL obj = new URL(url);
            HttpURLConnection con = (HttpURLConnection) obj.openConnection();
            con.setRequestMethod("GET");
            con.setRequestProperty("apikey", apiKey);

            int responseCode = con.getResponseCode();
            System.out.println("Response Code : " + responseCode);

            BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
            String inputLine;
            StringBuffer response = new StringBuffer();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();

            System.out.println(response.toString());
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

Listing 1: Ejemplo de consumo de API con Java

Este código en Java utiliza la biblioteca HttpURLConnection para realizar una solicitud GET a la API

de apilayer.com. En la solicitud, se especifica la URL de la API junto con la clave de API necesaria. La respuesta se recibe en formato JSON y se lee mediante un BufferedReader para almacenarla en un StringBuffer. Finalmente, la respuesta se imprime en la consola. Es importante señalar que se ha utilizado el bloque try-catch para manejar cualquier posible excepción que se produzca durante la solicitud de la API.

La salida del programa es:

```
Response Code : 200
{ "success": true, "query": { "from": "USD", "to": "EUR", "amount": 100 }, "info": { "timestamp": 1679928843,
     "rate": 0.92785 }, "date": "2023-03-27", "result": 92.785}
```

6. Bibliografía

Referencias

[1] Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[2] Platzi. *Curso de Java Básico*. Platzi, 2023.

https://platzi.com

[3] Platzi. *Curso de Java Orientado a Objetos*. Platzi, 2023.

https://platzi.com/

[4] Platzi. *Curso Avanzado de Java SE*. Platzi, 2023.

https://platzi.com

[5] Platzi. *Curso de Programación Orientado a Objetos*. Platzi, 2023.

https://platzi.com

[6] Platzi. *Curso Programación Funcional con Java*. Platzi, 2023.

https://platzi.com

[7] Oracle Corporation. *Java SE Documentation*. Oracle Corporation, 2023.

https://docs.oracle.com/en/java/javase/index.html