# Fundamentos de Java

1. ¿Qué es JDK, JRE y JVM?

   **JDK:** Java Development Kit es un software para los desarrolladores de Java. Incluye el intérprete Java, clases Java y herramientas de desarrollo Java (JDT): compilador, depurador, desensamblador, visor de applets, generador de archivos de apéndice y generador de archivos de apéndice y generador de documentación.

   Nos permite escribir aplicaciones que se desarrollan una sola vez y se ejecutan en cualquier lugar de cualquier máquina virtual Java. Las aplicaciones Java desarrolladas con el JDK en un sistema se pueden usar en otro sistema sin tener que cambiar ni recompilar el código. Los archivos de clase Java son portables a cualquier máquina virtual Java estándar.

   **JRE:** El entorno en tiempo de ejecución de Java es un software que los programas de Java necesitan para ejecutarse correctamente. El JRE es una tecnología subyacente que comunica el programa de Java con el sistema operativo. Actúa como traductor y facilitador, y brinda todos los recursos de modo que, una vez que escribe un software de java, se ejecuta en cualquier sistema operativo sin necesidad de más modificadores.

   **JVM:** La máquina virtual Java es un entorno de ejecución que puede añadir a un navegador o a cualquier sistema operativo. La máquina virtual Java ejecuta las instrucciones que genera un compilador Java. Consta de un intérprete de código de bytes y un tiempo de ejecución que permiten que los archivos de clase Java se ejecuten en cualquier plataforma, independientemente de la plataforma en la que se desarrollaron originalmente.

   Además de cargar y ejecutar los códigos de bytes, la máquina virtual Java incluye un recopilador de basura que gestiona la memoria.

2. ¿Cuáles son los tipos de dato primitivos, cuanta memoria ocupa cada uno y cuáles son sus rangos?
   - byte
     - Memoria: 1 byte (8 bits).
     - Rango: -128 a 127.
   - short
     - Memoria: 2 bytes (16 bits).
     - Rango: -32,768 a 32, 767
   - int
     - Memoria: 4 bytes (32 bits)
     - Rango: -2,147,483,648 a 2,147,483,647 ($-a\,2^{31}$ a $a\,2^{31} - 1$).

- long
  - Memoria: 8 bytes (64 bits).
  - Rango: -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 ($-2^{63}$ a $2^{63}-1$).
- float
  - Memoria: 4 bytes (32 bits).
  - Rango: Aproximadamente 1.4E-45 a 3.4E+38
- double
  - Memoria: 8 bytes (64 bits).
  - Rango: Aproximadamente 4.9E-324 a 1.7E+308
- boolean
  - Memoria: No está claramente definido, pero generalmente se usa 1 bit (depende de la implementación).
  - Rango: `true` o `false`
- char
  - Memoria: 2 bytes (16 bits).
  - Rango: (\u0000 a \uffff) 0 a 65,535 (representa un carácter Unicode).

3. ¿Qué es el casteo(casting) y para que se utiliza?
   Es una técnica utilizada en programación para convertir un valor de un tipo de dato a otro tipo de dato. Esto es útil cuando necesitas operar con diferentes tipos de datos y quieres asegurarte de que las operaciones se realicen correctamente. El casteo se utiliza para:
   - Convertir tipos de datos compatibles.
   - Mejorar la precisión o reducir la pérdida de información.
   - Compatibilidad de tipos en operaciones aritméticas.
   - Manejo de objetos en programación orientada a objetos.

4. ¿Qué es una clase y un objeto? (ejemplificar con un código breve y funcional)
   Una clase es un modelo o plantilla que define las propiedades (atributos) y comportamientos (métodos) de los objetos que se crean a partir de ella. En otras palabras, una clase es una descripción abstracta de algo que queremos representar en un programa, como una entidad con características y acciones.

   Un objeto, por otro lado, es una instancia de una clase. Es una representación concreta y específica de la clase que tiene valores específicos para los atributos definidos por la clase.

```java
public class Car {
    // Atributos de la clase
    String marca;
    String modelo;
    String color;
    int velocidad;

    // Constructor de la clase
    public Car(String marca, String modelo, String color) {
```

```
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.velocidad = 0; // Velocidad inicial es 0
    }

    // Método para acelerar el coche
    public void acelerar() {
        velocidad += 10;
        System.out.println("El coche ha acelerado. Velocidad actual: " +
velocidad + " km/h");
    }

    // Método para frenar el coche
    public void frenar() {
        if (velocidad >= 10) {
            velocidad -= 10;
        } else {
            velocidad = 0;
        }
        System.out.println("El coche ha frenado. Velocidad actual: " +
velocidad + " km/h");
    }
}



public class Main {
    public static void main(String[] args) {
        // Creación de un objeto de la clase Car
        Car miCoche = new Car("Toyota", "Camry", "Rojo");

        // Uso de los métodos del objeto
        miCoche.acelerar(); // Salida: El coche ha acelerado. Velocidad
actual: 10 km/h
        miCoche.acelerar(); // Salida: El coche ha acelerado. Velocidad
actual: 20 km/h
        miCoche.frenar();   // Salida: El coche ha frenado. Velocidad
actual: 10 km/h
    }
}
```

5. ¿Qué son las clases wrapper, para qué se usan y como se hacen las conversiones de datos primitivos a objetos mediante clases wrapper?

   Las clases wrapper (o clases envolventes) en Java son clases proporcionadas en el paquete java.lang que permiten utilizar tipos de datos primitivos (como int, char, boolean, etc.) como objetos. Cada tipo de dato primitivo tiene una clase wrapper correspondiente.

   Una clase wrapper es una clase que encapsula un tipo de dato primitivo en un objeto. Esto es útil cuando se necesita tratar los datos primitivos como objetos para utilizar características adicionales que los objetos permiten, como ser almacenados en colecciones (por ejemplo, en un ArrayList) o para realizar conversiones de datos.

Aquí están las clases wrapper correspondientes a cada tipo de dato primitivo:

- byte -> Byte
- short -> Short
- int -> Integer
- long -> Long
- float -> Float
- double -> Double
- char -> Character
- boolean -> Boolean

Las clases wrapper se usan para:

⬛ Almacenamiento en estructuras de datos de Java
⬛ Manipulación de datos como objetos
⬛ Manejo de null
⬛ Autoboxing y Unboxing

Las conversiones de datos primitivos a objetos mediante clases wrapper se hace de la siguiente forma:

**Autoboxing (Conversión automática de tipos primitivos a objetos):**

Autoboxing ocurre automáticamente cuando se asigna un valor primitivo a una variable de su clase wrapper correspondiente.

En este ejemplo, el tipo primitivo int se convierte automáticamente a un objeto de tipo Integer.

```
int numeroPrimitivo = 5;
Integer numeroObjeto = numeroPrimitivo; // Autoboxing automático
```

**Unboxing (Conversión automática de objetos a tipos primitivos):**

Unboxing ocurre automáticamente cuando se asigna un objeto de una clase wrapper a una variable de su tipo primitivo correspondiente.

En este ejemplo, el objeto Integer se convierte automáticamente a un tipo primitivo int.

```
Integer numeroObjeto = 10;
int numeroPrimitivo = numeroObjeto; // Unboxing automático
```

Aparte de autoboxing y unboxing, también puedes realizar conversiones explícitas usando métodos específicos proporcionados por las clases wrapper.

**Conversión de tipo primitivo a objeto:**

```
int numeroPrimitivo = 42;
Integer numeroObjeto = Integer.valueOf(numeroPrimitivo); // Conversión
explícita
```

**Conversión de objeto a tipo primitivo:**

```
Integer numeroObjeto = 100;
int numeroPrimitivo = numeroObjeto.intValue(); // Conversión explícita
```

6. ¿Cuál es la diferencia al almacenar en la memoria una variable local y una variable tipo objeto? ¿Para qué sirve el garbage colector?

   Diferencias al almacenar en memoria:

   **Variables locales.**

   **Ubicación en Memoria**: Las variables locales se almacenan en la pila de ejecución (stack). La pila es una sección de la memoria que se utiliza para almacenar variables locales y parámetros de métodos que se crean durante la ejecución de un método.

   **Tiempo de Vida**: Las variables locales existen únicamente durante la ejecución del método en el que fueron declaradas. Una vez que el método termina su ejecución, todas las variables locales de ese método se eliminan automáticamente de la pila.

   **Acceso Rápido**: Acceder a variables locales es muy rápido, ya que la pila de ejecución es muy eficiente para la gestión de memoria debido a su estructura LIFO (Last In, First Out).

   En conclusión, la diferencia al almacenar en la memoria una variable local y una variable de tipo objeto radica en cómo se gestionan en la memoria, su ciclo de vida y el uso del garbage collector (recolector de basura).

   **Variables de tipo Objeto.**

   **Ubicación en Memoria**: Las variables de tipo objeto, también conocidas como referencias a objetos, se almacenan en la pila, pero el objeto real al que apuntan se almacena en el montículo de memoria (heap). El montículo es una sección más grande de la memoria utilizada para el almacenamiento dinámico de objetos.

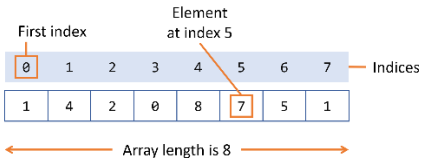
   **Tiempo de Vida**: Los objetos en el montículo tienen un tiempo de vida más largo que las variables locales. Un objeto permanece en el montículo mientras haya una referencia activa a él. Si ya no hay referencias al objeto, se considera "alcanzable para la recolección de basura".

   **Acceso Relativamente Más Lento**: Acceder a los objetos en el montículo es más lento que acceder a las variables en la pila debido a la sobrecarga de administrar la memoria dinámica.

   El **garbage collector (GC)** o **recolector de basura** es una herramienta automática que administra la memoria en lenguajes de programación como Java. Su función principal es liberar memoria en el montículo que ya no está siendo utilizada por el programa, es decir, memoria que está ocupada por objetos a los que ya no se hace referencia.

7. ¿Qué son los arreglos y como se utilizan?

   Un arreglo es un objeto contenedor que almacena un numero fijo de valores de un solo tipo. El tamaño del arreglo es establecido cuando el arreglo es creado. Después de la creación del arreglo el tamaño se fija al establecido.




El como se utiliza pues puede variar dependiendo de lo que queremos hacer, aquí un ejemplo genérico:

```
class ArrayDemo {
    public static void main(String[] args) {
        // Se declara un arreglo de enteros
        int[] anArray;

        // Se reserva memoria para el arreglo
        anArray = new int[10];

        // Inicialización del primer elemento.
        anArray[0] = 100;
        // Inicialización del Segundo elemento
        anArray[1] = 200;
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                           + anArray[0]);
        System.out.println("Element at index 1: "
                           + anArray[1]);
        System.out.println("Element at index 2: "
                           + anArray[2]);
        System.out.println("Element at index 3: "
                           + anArray[3]);
        System.out.println("Element at index 4: "
                           + anArray[4]);
        System.out.println("Element at index 5: "
                           + anArray[5]);
        System.out.println("Element at index 6: "
                           + anArray[6]);
        System.out.println("Element at index 7: "
                           + anArray[7]);
        System.out.println("Element at index 8: "
                           + anArray[8]);
        System.out.println("Element at index 9: "
                           + anArray[9]);
    }
}
```

Podemos observar la forma en la que podemos declarar y llenar un arreglos sin embargo la mejor forma de hacerlo es por medio de un ciclo for y usando un contador para controlar el index.

8. ¿Cómo se le pueden pasar argumentos al programa mediante el arreglo `String[] args`?

   •Los argumentos se pasan como cadenas (String), por lo que si necesitas un número u otro tipo de dato, debes convertirlos explícitamente, como se hizo con Integer.parseInt(args[1]).

   • Siempre es buena práctica verificar la longitud de args antes de intentar acceder a sus elementos para evitar errores.

   • Número de argumentos: Puedes verificar el número de argumentos pasados usando args.length. Si args.length es 0, significa que no se han pasado argumentos.

   • Acceso a argumentos específicos: Puedes acceder a cualquier argumento usando su índice, por ejemplo, args[0] para el primer argumento, args[1] para el segundo, y así sucesivamente.

   • Conversión de tipo: Todos los argumentos en args son de tipo String. Si necesitas usarlos como otros tipos de datos, como int o double, debes convertirlos utilizando métodos como Integer.parseInt() o Double.parseDouble().

9. ¿Qué es un package, para que se usan y como se importan?

   **¿Qué es?**

   Un paquete es un espacio de nombres que organiza un conjunto de clases e interfaces relacionadas. Conceptualmente, puedes pensar que los paquetes son similares a diferentes carpetas en tu computadora. Puede mantener páginas HTML en una carpeta, imágenes en otra y scripts o aplicaciones en otra más.

   La plataforma Java proporciona una enorme biblioteca de clases (un conjunto de paquetes) adecuada para usar en sus propias aplicaciones.

   Esta biblioteca se conoce como "Interfaz de programación de aplicaciones" o "API" para abreviar. Sus paquetes representan las tareas más comúnmente asociadas con la programación de propósito general.

   **¿Para qué se usan?**

   •Organización del código: Permite organizar clases e interfaces relacionadas en un mismo grupo, mejorando la legibilidad y mantenimiento del código.

   •Evitar conflictos de nombres: Diferentes desarrolladores pueden crear clases con el mismo nombre sin que haya conflictos, siempre y cuando estén en paquetes diferentes.

   •Control de acceso: Los paquetes pueden ayudar a controlar el acceso a clases e interfaces mediante modificadores de acceso como public, protected, private, y el acceso por defecto (package-private).

   **¿Cómo se importan?**

   Se utiliza la palabra clave import para acceder a clases de otros paquetes.

10. ¿Qué es la clase String y cuáles son sus métodos más importantes?

    La clase **String** en Java es una clase final e inmutable que se utiliza para representar cadenas de texto (secuencias de caracteres). Una vez que se crea un objeto String, su contenido no puede ser modificado. Las cadenas en Java se manejan de forma diferente a

otros lenguajes porque la clase String ofrece una variedad de métodos útiles para manipular y operar con texto. Sus metoos mas importantes son:

**1.length()**:

- **Descripción**: Devuelve la longitud de la cadena.
- **Uso:**

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

**2.charAt(int index)**:

- **Descripción**: Devuelve el carácter en la posición especificada.
- **Uso**:

```
String str = "Hola";
char c = str.charAt(1); // Devuelve 'o'
```

**3.substring(int beginIndex)** y **substring(int beginIndex, int endIndex)**:

- **Descripción**: Devuelve una nueva cadena que es una subcadena de la cadena original.
- **Uso:**

```
String str = "Hola Mundo";
String subStr1 = str.substring(5); // Devuelve "Mundo"
String subStr2 = str.substring(0, 4); // Devuelve "Hola"
```

**4.indexOf(String str)** y **indexOf(String str, int fromIndex)**:

- **Descripción**: Devuelve el índice de la primera aparición de la subcadena especificada, o -1 si no se encuentra.
- **Uso:**

```
String str = "Hola Mundo";
int index1 = str.indexOf("Mundo"); // Devuelve 5
int index2 = str.indexOf("a", 2);  // Devuelve 3
```

11. ¿Para qué se usa la palabra reservada this?

    La palabra reservada this en Java se utiliza dentro de un método o constructor para referirse al objeto actual de la clase en la que se está escribiendo. Es una referencia al propio objeto en el contexto de su clase y tiene varios usos importantes:

    1. Diferenciar entre los atributos de instancia y los parámetros o constructor.
    2. Invocar otro constructor de la misma clase.
    3. Pasar el objeto actual como un argumento al método o constructor de otra clase.
    4. Retornar el objeto actual desde un método.
    5. Evitar el uso de referencias globales o estáticas.

12. ¿Qué es la herencia? (ejemplificar con un código breve y funcional)

    La **herencia** en Java es un concepto fundamental de la programación orientada a objetos (OOP) que permite a una clase (denominada **subclase** o **clase derivada**) heredar campos y

métodos de otra clase (denominada **superclase** o **clase base**). Esto facilita la reutilización del código y la creación de relaciones jerárquicas entre clases.

```
// Superclase Vehiculo
public class Vehiculo {
    protected String marca;
    protected int anio;

    public Vehiculo(String marca, int anio) {
        this.marca = marca;
        this.anio = anio;
    }

    public void mostrarInfo() {
        System.out.println("Marca: " + marca + ", Año: " + anio);
    }
}

// Subclase Coche que hereda de Vehiculo
public class Coche extends Vehiculo {
    private int puertas;

    public Coche(String marca, int anio, int puertas) {
        super(marca, anio); // Llamada al constructor de la superclase
        this.puertas = puertas;
    }

    public void mostrarInfo() {
        super.mostrarInfo(); // Llamada al método de la superclase
        System.out.println("Número de puertas: " + puertas);
    }

    public void tocarBocina() {
        System.out.println("¡Beep beep!");
    }
}

// Clase principal para probar la herencia
public class Main {
    public static void main(String[] args) {
        Vehiculo miVehiculo = new Vehiculo("Toyota", 2010);
        miVehiculo.mostrarInfo(); // Marca: Toyota, Año: 2010

        Coche miCoche = new Coche("Honda", 2022, 4);
        miCoche.mostrarInfo(); // Marca: Honda, Año: 2022, Número de
puertas: 4
        miCoche.tocarBocina();  // ¡Beep beep!
    }
}
```

13. ¿Qué es el polimorfismo? (ejemplificar con un código breve y funcional)

    El **polimorfismo** es uno de los principios fundamentales de la programación orientada a objetos (OOP). En términos simples, el polimorfismo permite que un objeto adopte muchas formas. En Java, el polimorfismo se refiere a la capacidad de una variable, objeto o método de tomar múltiples formas. Más específicamente, en Java, existen dos tipos principales de polimorfismo:

    1. **Polimorfismo en tiempo de compilación (sobrecarga de métodos)**: Ocurre cuando varios métodos en la misma clase tienen el mismo nombre pero diferentes parámetros (diferente número o tipo de argumentos).
    2. **Polimorfismo en tiempo de ejecución (sobreescritura de métodos)**: Ocurre cuando una subclase proporciona una implementación específica de un método que ya está definido en su superclase. La referencia de un objeto puede ser de la superclase, pero el método que se ejecuta es el de la subclase.

    **Ejemplo de polimorfismo en tiempo de ejecución**.

```java
// Superclase Animal
public class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }
}

// Subclase Perro que sobrescribe el método hacerSonido
public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra");
    }
}

// Subclase Gato que sobrescribe el método hacerSonido
public class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El gato maúlla");
    }
}

// Clase principal para demostrar el polimorfismo
public class Main {
    public static void main(String[] args) {
        // Crear objetos de tipo Animal pero con diferentes
implementaciones
        Animal miAnimal = new Animal();
        Animal miPerro = new Perro();
        Animal miGato = new Gato();

        // Llamar al método hacerSonido() en cada objeto
        miAnimal.hacerSonido();  // Salida: El animal hace un sonido
        miPerro.hacerSonido();   // Salida: El perro ladra
        miGato.hacerSonido();    // Salida: El gato maúlla
    }
}
```

**Ejemplo de polimorfismo en tiempo de compilación.**

```java
public class Calculadora {
    // Método suma para dos enteros
    public int suma(int a, int b) {
        return a + b;
    }

    // Método suma para tres enteros
    public int suma(int a, int b, int c) {
        return a + b + c;
    }

    // Método suma para dos números en punto flotante
    public double suma(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        // Llamadas a los métodos sobrecargados
        System.out.println(calc.suma(5, 10));          // Salida: 15
        System.out.println(calc.suma(5, 10, 15));      // Salida: 30
        System.out.println(calc.suma(5.5, 10.2));      // Salida: 15.7
    }
}
```

14. ¿Qué es el @Override y para que sirven los métodos `toString()` y `equals()`?

    **¿Qué es el @Override?**

    El @Override es una anotación en Java que se utiliza para indicar que un método en una clase está sobrescribiendo (overriding) un método en su superclase. Esta anotación no es obligatoria, pero es una buena práctica utilizarla porque proporciona varios beneficios:

1. Verificación en tiempo de compilación: Cuando utilizas @Override, el compilador de Java verifica que realmente estás sobrescribiendo un método de la superclase. Si el método en la subclase no coincide con el método de la superclase en la firma (nombre, parámetros y tipo de retorno), el compilador generará un error. Esto ayuda a evitar errores comunes, como errores tipográficos o de firma incorrecta.
2. Mejor legibilidad del código: Utilizar @Override mejora la legibilidad del código, indicando claramente que un método está sobrescribiendo un método en la superclase.

    **¿Para qué sirve toString?**

    El método toString() es un método de la clase Object en Java, que es la superclase de todas las clases en Java. Este método devuelve una representación de cadena del objeto. De forma predeterminada, toString() devuelve una cadena que consiste en el nombre de la clase, seguido de un signo @, y luego el valor hexadecimal del código hash del objeto. Sin embargo, este comportamiento predeterminado no es muy útil, por lo que generalmente se sobrescribe para proporcionar una representación más significativa del objeto.

**¿Para qué sirve equals?**

El método equals() también es un método de la clase Object que se utiliza para comparar si dos objetos son iguales. De forma predeterminada, el método equals() en Object simplemente verifica si dos referencias de objeto apuntan al mismo objeto en memoria (es decir, compara las referencias de objeto, no los valores).

Sin embargo, este comportamiento predeterminado rara vez es lo que se desea. La mayoría de las veces, queremos comparar el contenido de dos objetos (es decir, comparar los valores de sus atributos). Para lograr esto, sobrescribimos el método equals().

15. ¿Qué es la sobrecarga de métodos? (ejemplificar con un código breve y funcional)

    La **sobrecarga de métodos** (method overloading) en Java es una característica que permite a una clase tener más de un método con el mismo nombre, siempre y cuando cada uno tenga un conjunto diferente de parámetros. Esto significa que puedes definir múltiples versiones de un método dentro de la misma clase, pero cada versión debe diferir en el número de parámetros, los tipos de parámetros, o ambos.

```java
public class Calculadora {
    // Método suma para dos números enteros
    public int suma(int a, int b) {
        return a + b;
    }

    // Método suma para tres números enteros
    public int suma(int a, int b, int c) {
        return a + b + c;
    }

    // Método suma para dos números de tipo double
    public double suma(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        // Llamadas a los métodos sobrecargados
        System.out.println(calc.suma(5, 10));         // Salida: 15
        System.out.println(calc.suma(5, 10, 15));     // Salida: 30
        System.out.println(calc.suma(5.5, 10.2));     // Salida: 15.7
    }
}
```

16. ¿Qué es el manejo de excepciones? (ejemplificar con un código breve y funcional)

    El **manejo de excepciones** en Java es un mecanismo que permite manejar de manera controlada los errores y situaciones excepcionales que ocurren durante la ejecución de un programa. Una **excepción** es un evento que interrumpe el flujo normal de un programa y puede ser causado por diversas razones, como errores de programación, recursos no disponibles o datos incorrectos.

```java
public class ManejoExcepciones {
    public static void main(String[] args) {
        try {
            // Código que puede lanzar una excepción
            int numerador = 10;
            int denominador = 0;
            int resultado = dividir(numerador, denominador);
            System.out.println("Resultado: " + resultado);
        } catch (ArithmeticException e) {
            // Manejo de la excepción
            System.out.println("Error: No se puede dividir por cero.");
        } finally {
            // Código que siempre se ejecuta
            System.out.println("Operación de división finalizada.");
        }
    }

    public static int dividir(int numerador, int denominador) {
        return numerador / denominador; // Puede lanzar
ArithmeticException
    }
}
```

17. ¿Para que se utiliza `static`? (véase: https://www.youtube.com/watch?v=mvBX4-5-A4o )

La palabra clave **static** en Java se utiliza para definir miembros de una clase (métodos, variables, bloques, e incluso clases internas) que pertenecen a la clase misma en lugar de a instancias individuales de la clase. Esto significa que los miembros estáticos se comparten entre todas las instancias de una clase y se pueden acceder directamente a través de la clase sin necesidad de crear un objeto de la clase.

**Usos de static**

1. **Variables estáticas**: También conocidas como **variables de clase**, son variables que se comparten entre todas las instancias de una clase. Si se modifica una variable estática en una instancia, el cambio se refleja en todas las instancias de esa clase.
2. **Métodos estáticos**: Los **métodos estáticos** se pueden llamar sin crear una instancia de la clase. No pueden acceder directamente a miembros de instancia (variables no estáticas y métodos no estáticos) porque los miembros estáticos no pertenecen a ninguna instancia.
3. **Bloques estáticos**: Son bloques de código que se ejecutan una vez cuando la clase se carga en la memoria, antes de que se creen objetos de la clase o se invoquen métodos estáticos.
4. **Clases internas estáticas**: Una clase interna que es estática se puede crear sin una instancia de la clase externa. A menudo se utiliza para definir clases utilitarias o constantes.

Subir el documento terminado a teams en formato pdf para acreditar la presente tarea.