Módulo 3. POO: clases abstractas y estáticas, interface y estructuras de control

CÓRDOBA CREA FUTURO

1. Pilares de la POO

Los cuatro pilares de la programación orientada a objetos (POO) son la abstracción, la encapsulación, la herencia y el polimorfismo. Estos principios ayudan a diseñar programas más eficientes, flexibles y fáciles de mantener al permitir ocultar detalles, reutilizar código y gestionar objetos de forma más sencilla.

Figura 1: Pilares de la programación orientada a objetos (POO)

Fuente: elaboración propia.

Abstracción

Se centra en las características esenciales de un objeto, ocultando la complejidad interna e irrelevante. Permite al usuario interactuar con un objeto sin necesidad de conocer todos sus detalles de funcionamiento interno.

Este pilar se refiere a la capacidad de representar las características esenciales de un objeto sin incluir detalles innecesarios. Permite centrarse en lo que un objeto hace en lugar de cómo lo hace.

Ejemplo: una clase Vehiculo puede abstraer atributos como velocidadMaxima y métodos como acelerar(), sin definir cómo se implementan; solo se definen.

~~Código de ejemplo Java~~

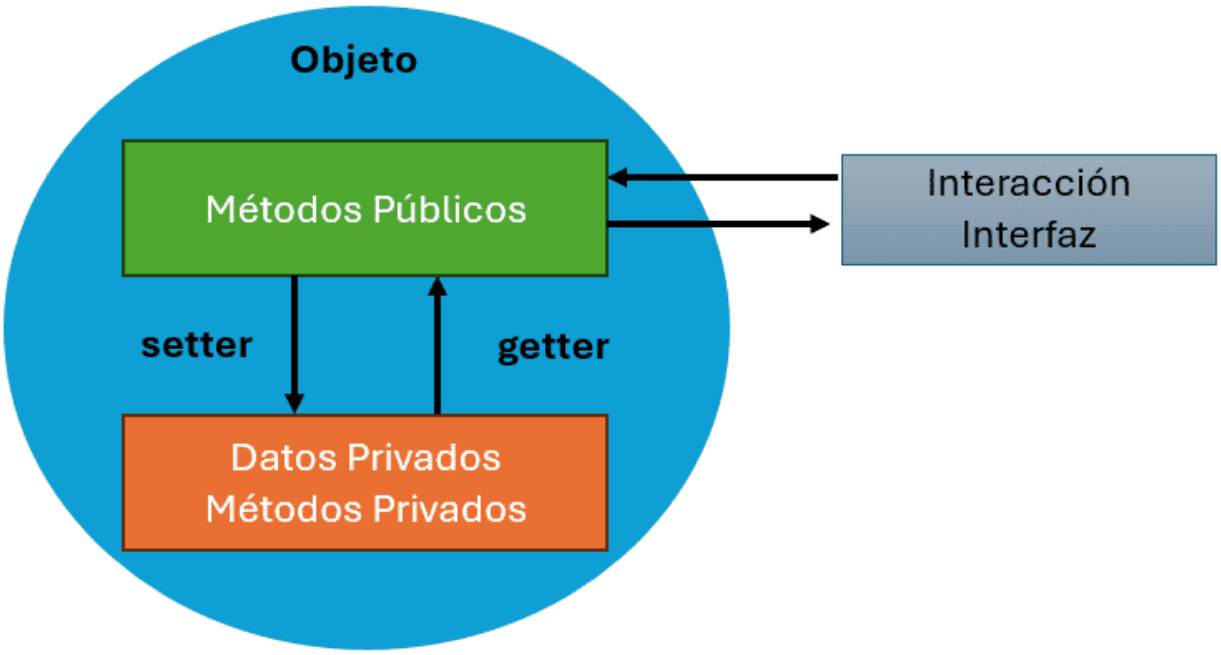
```
// Nivel de abstracción - interfaz

interface Reproductor {

  void reproducir();

  void pausar();

  void detener();

}
```

Se puede acceder a un ejemplo completo y ejecutarlo haciendo clic en este enlace: https://www.online-java.com/SJ2pwSFkzI.

Encapsulación

Consiste en agrupar los datos (atributos) y los métodos que operan sobre esos datos dentro de una sola unidad (la clase), controlando el acceso a ellos y protegiendo los datos de modificaciones externas no deseadas, es decir, oculta la implementación interna, mostrando solo una interfaz pública.

Figura 2: El objeto se comunica con el exterior por métodos o atributos públicos



Fuente: elaboración propia.

Algunos atributos o métodos pueden ser definidos como privados para que desde el exterior al objeto no puedan ser modificados directamente, pero pueden ser accedidos (*getters*) o modificados (*setter*) desde el propio objeto. Esto protege los datos de accesos no autorizados y facilita la modificación del código interno sin afectar otras partes del programa.

Ejemplo: previene modificaciones inadecuadas (por ejemplo, atributos privados con *getters*/*setters*).

Ejecuta el siguiente código de ejemplo haciendo clic en el siguiente enlace: https://www.online-java.com/J2lMa648Td.

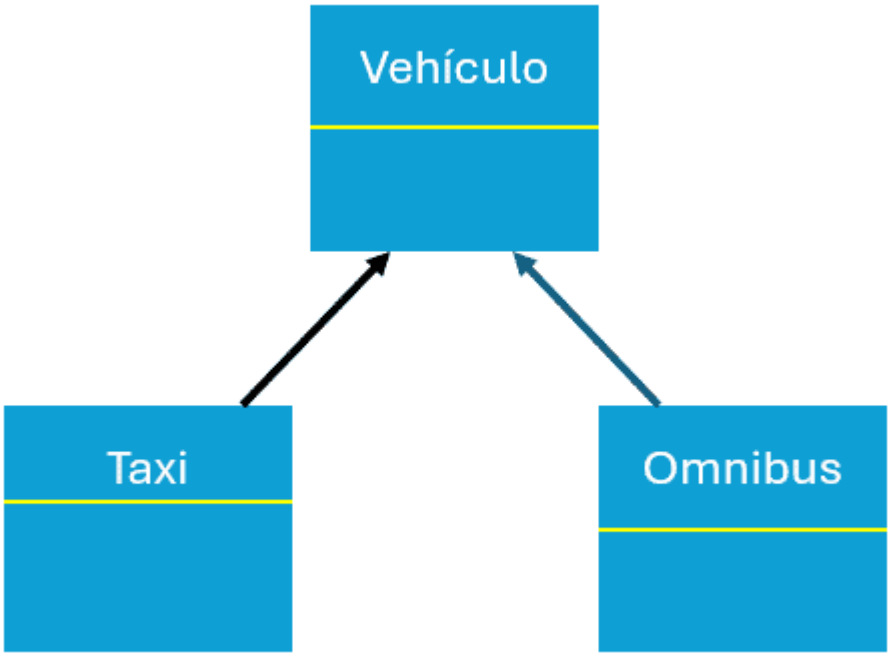
Herencia

La herencia permite que una clase (subclase o clase hija) herede atributos y métodos de otra clase (superclase o clase

padre). Esto promueve la reutilización de código y establece relaciones jerárquicas entre clases.

La forma de expresarlo sería que la clase hija se extiende de la clase padre.

Figura 3: La clase Taxi extends Vehiculo



Fuente: elaboración propia.

Para ver un ejemplo funcional de herencia, se invita al alumno a analizar el siguiente código haciendo clic sobre el siguiente enlace: https://www.online-java.com/4MO8LwmoCf.

El alumno podrá usar cualquier *prompt* de IA y solicitar que le hagan un resumen copiándolo y pegándolo.

Polimorfismo

Proviene de las palabras griegas «*poli*» (muchos) y «*morfos*» (formas), es decir, «muchas formas». Permite que objetos de diferentes clases respondan de la misma manera a la misma llamada a un método, aunque la implementación específica de ese método pueda ser diferente para cada clase. Un ejemplo sería decir algo así como indicar bailar mencionando a varias personas; es la misma orden, pero cada uno baila a su forma.

Figura 4: Pilares de la POO

Fuente: elaboración propia.

Capacidad de que un objeto se comporte de múltiples formas

Se mencionan dos tipos de polimorfismos:

- a) polimorfismo de subtipo (sobrescrita de métodos). Una subclase redefine un método de su superclase.
- b) Polimorfismo paramétrico (sobrecarga): múltiples métodos con el mismo nombre, pero distintos parámetros.

A continuación, se presenta un ejemplo usando código fuente parcial de Java, es decir, no está completo, pero se puede apreciar:

```
// INTERFACE (contrato común)

        interface Forma {

          double calcularArea();

          void dibujar();
```

```
}

// DIFERENTES IMPLEMENTACIONES

class Circulo implements Forma {

  private double radio;

    // sobre escribo dibujar y calcularArea

  @Override

  public void dibujar() {

     System.out.println("Dibujando un círculo ⭕");

  }

  @Override

  public double calcularArea() {

     return Math.PI radio radio;

  }
```

}

CONTINUAR

2. Constructor

Un constructor en programación orientada a objetos (POO) es un método especial que se usa para crear e inicializar un objeto de una clase. Se invoca automáticamente cuando se crea una nueva instancia de la clase y su objetivo principal es asignar los valores iniciales a los atributos del objeto para que esté listo para su uso. Un constructor tiene el mismo nombre que la clase; se puede tener varios constructores, pero con diferentes cantidades de parámetros, aplicando el concepto de polimorfismo, como se muestra en el siguiente ejemplo:

```
public class Persona {

    private String nombre;

    private int edad;

    // CONSTRUCTOR con dos parámetros, tiene el mismo nombre que la clase
```

```
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
// CONSTRUCTOR con un (1) parámetro, tiene el mismo nombre que la clase
    public Persona(String nombre) {
        this.nombre = nombre;
        this.edad = 21;
    }
 }
public Main {
   public static void main(String[] args) {
```

```
    // Invoco al CONSTRUCTOR con dos parámetros y luego con uno solo

    Persona persona  = New Persona(“Walter”, 25);

    Persona persona1 = New Persona(“Fabian”);

  }

}
```

El siguiente ejemplo muestra un pequeño programa en el que se puede ver la ejecución y acción del constructor: https://www.online-java.com/KM4Z9yPvB0.

CONTINUAR

3. Interface

En la programación orientada a objetos (POO) de Java, las interfaces en Java son una parte crucial de la POO que permiten definir un contrato que las clases deben cumplir. En esencia, una interfaz en Java define un conjunto de métodos que deben ser implementados por cualquier clase que aplique esa interfaz. Es como un acuerdo formal que garantiza que una clase tendrá ciertos comportamientos.

La principal característica de una interfaz es que solo declara métodos, pero no proporciona aplicaciones para ellos. Es decir, no define el **cómo** de la funcionalidad, solo especifica el **qué**. Las clases que aplican una interfaz deben proporcionar sus propias implementaciones para cada uno de los métodos declarados en la interfaz.

Declaración de interfaces en Java

La declaración de una interfaz en Java es sencilla y sigue una sintaxis específica. Para declarar una interfaz, usamos la palabra clave interface, seguida del nombre de la interfaz y un bloque de código que contiene la lista de métodos que la interfaz va a declarar.

```
public interface MiInterfaz {

  // Declaración de métodos (sin implementación), solo se mencionan

 void metodo1();

 int metodo2(String parametro); }
```

En este ejemplo, usamos la palabra clave interface para declarar la interfaz MiInterfaz. La interfaz no contiene aplicaciones de métodos, solo declara los nombres y las firmas de los métodos (metodo1 y metodo2, en este caso).

Las interfaces también pueden contener constantes, las cuales son implícitamente public, static y final.

```
public interface OtraInterfaz {
```

```
// Declaración de constantes

int CONSTANTE1 = 42;

String CONSTANTE2 = "Hola";

// Declaración de métodos (sin implementación)

void metodo(); }
```

Implementación de interfaces en clases

Para aplicar una interfaz en una clase, se usa la palabra clave **implements**. La clase que aplica una interfaz debe proporcionar implementaciones para todos los métodos declarados en la interfaz.

Supongamos que tenemos una interfaz llamada MiInterfaz con dos métodos (metodo1 y metodo2), para crear una clase llamada MiClase que implementa esta interfaz procederíamos de la siguiente manera:

```
public class MiClase implements MiInterfaz {

    @Override
```

```
    public void metodo1() {
        // Implementación del método1
        System.out.println("Implementación de método1");
    }

    @Override
    public int metodo2(String parametro) {
        // Implementación del método2
        return parametro.length();
    }
}
```

Se ha usado la palabra clave **implements** para indicar que la clase implementa la interfaz MiInterfaz; además, la clase proporciona implementaciones para los métodos definidos en la interfaz.

Ejemplos prácticos de uso de interfaces

En el siguiente ejemplo, se muestra cómo podríamos implementar más de una interfaz en una clase:

```
interfaceInterfazA {

  voidmetodoA();

}

 interfaceInterfazB {

  voidmetodoB();

}

classMiClaseimplementsInterfazA, InterfazB {

  @Override

  publicvoidmetodoA() {

     System.out.println("Implementación de metodoA");

  }    @Override

  publicvoidmetodoB() {
```

```
        System.out.println("Implementación de metodoB");

    }

}
```

Supongamos ahora que tienes diferentes tipos de vehículos y que quieres que todos puedan arrancar y detenerse. Puedes usar una interfaz para esto:

```
interface Vehiculo {

    void arrancar();

    void detener();

}

class Automovil implements Vehiculo {

    @Override    public void arrancar() {

        System.out.println("Arrancando automóvil...");

    }
```

```
    @Override
    public void detener() {
        System.out.println("Deteniendo automóvil...");
    }
} class Motocicleta implements Vehiculo {
    @Override
    public void arrancar() {
        System.out.println("Arrancando motocicleta...");
    }
    @Override
    public void detener() {
        System.out.println("Deteniendo motocicleta...");
    }
```

```
}
```

CONTINUAR

4. Paquetes

Un paquete (también conocido como *package* en inglés) es un mecanismo que se usa para organizar y estructurar clases relacionadas y otros elementos dentro de un proyecto. Los paquetes ayudan a evitar conflictos de nombres, proporcionan una estructura de directorios de lógica y permiten el control de acceso a clases y miembros.

Un paquete agrupa clases, interfaces, enumeraciones y subpaquetes relacionados. Cada clase en Java debe pertenecer a un paquete, ya sea explícitamente especificado o, formando parte del paquete predeterminado.

Organización de clases en paquetes

La organización de clases en paquetes es una práctica común para mantener un código bien estructurado y modular. Los paquetes ayudan a organizar y agrupar clases

relacionadas en un espacio de nombres específico, ¿pero cómo organizamos clases en paquetes?

Crea un paquete

Para organizar clases en un paquete, primero debes crear el paquete. Un paquete es simplemente un directorio en el sistema de archivos que contiene las clases relacionadas.

Estructura las carpetas

Debemos asegurarnos de que la estructura refleje la jerarquía de los paquetes. Por ejemplo:

Importación de clases de otros paquetes

La importación de clases de otros paquetes en Java permite usar esas clases en nuestro código actual; hay diferentes maneras de hacer esto.

Se puede importar una clase específica de otro paquete usando la palabra clave **import** seguida del nombre completo de la clase. Por ejemplo:

```
import com.otropaquete.OtraClase;

    public class MiClase {

        OtraClase otraClase; // Puedes usar OtraClase en esta clase

    }
```

Si deseas importar todas las clases de un paquete, puedes usar un comodín *. Esto importará todas las clases del paquete. Por ejemplo:

```
import com.otropaquete.*;
```

También puedes importar miembros estáticos (como variables o métodos estáticos) de una clase en otro paquete de forma estática. Esto se hace usando la palabra clave **import static**. Por ejemplo:

```
import static com.otropaquete.ClaseEstatica.metodoEstatico;
```

```
public class MiClase {

    public void miMetodo() {

        int resultado = metodoEstatico(10, 20);

        // Usar el método estático de ClaseEstatica

    }

}
```

CONTINUAR

5. Clase abstracta y estática

Una **clase abstracta** es un molde incompleto que no se puede instanciar directamente y se usa para definir un «esqueleto» común para otras clases, que deben aplicar sus métodos abstractos (sin cuerpo). Una **clase estática** no se puede instanciar y se usa principalmente para agrupar métodos y variables utilitarios que se pueden usar directamente desde la clase, sin crear un objeto de ella.

Clase abstracta

- **Propósito**: sirve como una clase base que define un contrato común para un conjunto de clases relacionadas.
- **No instanciable**: no puedes crear un objeto (instancia) a partir de una clase abstracta.

- **Métodos abstractos**: puede contener métodos abstractos, los cuales tienen una declaración (firma), pero no una implementación. Las subclases que heredan de ella deben obligatoriamente implementar estos métodos.

- **Métodos concretos**: también puede tener métodos con código, que son compartidos por todas sus subclases.

- **Herencia**: una clase puede heredar de una única clase abstracta.

Clase estática

- **Propósito**: agrupar funcionalidades que no dependen de un estado de instancia particular. Por ejemplo, los métodos de la clase Math de Java.

- **No instanciable**: una clase estática no se puede instanciar.

- **Miembros estáticos**: contiene principalmente métodos y variables estáticos que se acceden directamente usando el nombre de la clase (por ejemplo, NombreClase.metodo()).

- **Función utilitaria**: se usa comúnmente para crear utilidades o *helpers*, en los que la lógica no necesita ser parte de un objeto específico.

Diferencia clave

La diferencia principal radica en su propósito y estructura: la clase abstracta se centra en la **herencia y la abstracción de comportamiento**, definiendo un "es-un" (por ejemplo, Animal es-un Mamifero) y requiriendo implementación de métodos, mientras que la clase estática se centra en la **agrupación de utilidades** que no requieren un objeto para ser usadas. Una clase abstracta puede ser un «esqueleto» que se completa en las subclases, mientras que una clase estática es un conjunto de herramientas que se usan directamente sin necesidad de crear un objeto.

CONTINUAR

6. Estructuras de control

Tal como se vio en la lectura 1, las estructuras de control son del mismo tipo en todos los lenguajes en los que suelen variar las anotaciones; por ejemplo, en Java se usan los paréntesis para indicar el bloque de sentencias a ejecutar, pero otros lenguajes pueden tener anotaciones o palabras reservadas distintas.

Las estructuras de control en Java son la forma de gestionar el flujo de ejecución de un programa y se clasifican en selectivas (condicionales), repetitivas (bucles) y de salto. Las estructuras repetitivas (como for, while, do-while) ejecutan un bloque de código múltiples veces. Finalmente, las estructuras de salto (break, continue, return) se usan para alterar el curso normal de ejecución dentro de los bucles o métodos.

Condicional

Las estructuras selectivas (como if, else if, switch) permiten ejecutar bloques de código diferentes dependiendo del hecho de que una condición sea verdadera o falsa.

If

La declaración if-else es la más básica de todas las estructuras de control y es probablemente también la declaración de toma de decisiones más común en la programación.

Nos permite ejecutar una sección de código específica solo si se cumple una condición específica.

La declaración **if** siempre necesita una expresión booleana como su parámetro.

Figura 5: Ejemplo

```
if (condición) {
   // Se ejecuta cuando la condición es verdadera.
} else {
   // Se ejecuta cuando la condición es falsa.
}
```

Fuente: elaboración propia.

Figura 6: Ejemplo

```
Valor = 5;
if (valor > 1) {
    System.out.println("El valor es mayor que 1");
    System.out.println("El valor es igual a: " + valor);
}
```

Fuente: elaboración propia.

El mensaje "El valor es mayor que 1" y “El valor es igual a” solo se imprimirá si se cumple la condición.

Else

A continuación, podemos elegir entre dos acciones usando if y else juntos.

Figura 7: Ejemplo

```
if (valor > 2) {
    System.out.println("El valor es mayor que 2");
} else {
    System.out.println("El valor es menor o igual que 2");
}
```

Fuente: elaboración propia.

Else If

Figura 8: Ejemplo de sintaxis combinada if/else/else if

```java
if (i > 2) {
    System.out.println("i es mayor que 2");
} else if (i <= 0) {
    System.out.println("i es menor o igual a cero");
} else {
    System.out.println("i es igual a uno o dos");
}
```

Fuente: elaboración propia.

Switch

Si tenemos múltiples casos para elegir, podemos usar una declaración switch.

Figura 9: Switch

```
int cuenta = 3;

switch (cuenta) {

case 0:

    System.out.println("La cuenta es igual a 0");

    break;

case 1:

    System.out.println("La cuenta es igual a 1");

    break;

default:

    System.out.println("La cuenta es negativo o mayor que 1");

    break;

}
```

Fuente: elaboración propia.

Tabla 1: Ejemplo de if y switch

Sentencia IF / ELSE

```
public String ejemploDeIf(String animal) {
   String resultado;
   if (animal.equals("PERRO") || animal.equals("GATO")) {
      resultado = "animal doméstico";
   } else if (animal.equals("TIGRE")) {
      resultado = "animal salvaje";
   } else {
      resultado = "animal desconocido";
   }
   return resultado;
}
```

Sentencia Switch

```
public String ejemploDeSwitch(String animal) {
   String resultado;
   switch (animal) {
      case "PERRO":
         resultado = "animal doméstico";
         break;
      case "GATO":
         resultado = "animal doméstico";
         break;
      case "TIGRE":
         resultado = "animal salvaje";
         break;
      default:
         resultado = "animal desconocido";
         break;
   }
   return resultado;
}
```

Fuente: elaboración propia.

Declaración break

Si olvidamos escribir **break**, se ejecutarán los bloques que estén debajo. Para demostrar esto, se omiten las declaraciones break y, luego de ejecutar el código que se muestra seguidamente, producirá una salida por consola así.

Figura 10: Salida por consola

```java
public String olvidarBreakEnSwitch(String animal) {
   switch (animal) {
   case "PERRO":
      System.out.println("animal doméstico");
   default:
      System.out.println("animal desconocido");
   }
}
```

Fuente: elaboración propia.

El resultado de la ejecución de este pedazo de código mostrará por consola el siguiente resultado:

animal doméstico

animal desconocido

Por lo tanto, debemos tener cuidado y agregar declaraciones break al final de cada bloque, a menos que sea necesario pasar al código bajo la siguiente etiqueta.

El único bloque en el que no es necesario un break es el último, pero agregar un break al último bloque hace que el código sea menos propenso a errores.

También podemos aprovechar este comportamiento para omitir el break, cuando deseamos que se ejecute el mismo

código para varios casos.

Figura 11: Omisión del break

```
public String ejemploDeSwitch(String animal) {
   String resultado;

   switch (animal) {
      case "PERRO":
      case "GATO":
         resultado = "animal doméstico";
         break;
      case "TIGRE":
         resultado = "animal salvaje";
         break;
      default:
         resultado = "animal desconocido";
         break;
   }
   return resultado;
}
```

Fuente: elaboración propia.

Bucles

Los bucles se usan cuando necesitamos repetir el mismo código varias veces consecutivas.

Veamos un ejemplo rápido de bucles comparables de tipo for y while.

Figura 12: For y while

```
BLOQUE FOR

for (int i = 1; i <= 50; i++) {
        metodoARepetir();
}


BLOQUE WHILE

int contadorWhile = 1;
while (contadorWhile <= 50) {
        metodoARepetir();
        contadorWhile++;
}
```

Fuente: elaboración propia.

Ambos bloques de código anteriores llamarán al metodoARepetir 50 veces.

Break en bucle

Necesitamos usar break para salir de un bucle antes de que finalice.

Veamos un ejemplo rápido.

Figura 13: Ejemplo

```
List<String> nombres = obtenerListaDeNombres();
String nombre = "John Doe";
int indice = 0;
for (; indice < nombres.length; indice++) {
   if (nombres[indice].equals(nombre)) {
      break;
   }
}
```

Fuente: elaboración propia.

Continue

En pocas palabras, continue significa saltar el resto del bucle en el que estamos.

Figura 14: Continue

```java
List<String> nombres = obtenerListaDeNombres();
String nombre = "John Doe";
String lista = "";
for (int i = 0; i < nombres.length; i++) {
   if (nombres[i].equals(nombre)) {
      continue;
   }
   lista += nombres[i];
}
```

Fuente: elaboración propia.

Aquí, omitimos agregar los nombres duplicados a la lista.

CONTINUAR