

**INFORME DEL MODELO CONCEPTUAL ORIENTADO A OBJETOS**

**PRESENTADO A:**

**INST. ANDRÉS FERNANDO SÁNCHEZ SOLARTE**

**PRESENTADO POR:**

**CARLOS JOSÉ DELGADO GONZÁLEZ**

**DURLEY SANDRITH GALVÁN JIMÉNEZ**

**YASSER LEONARDO PACHECO CAÑIZARES**

**JORGE ANDERSON CORTÉS TORRES**

**PROGRAMA:**

**PROGRAMACION DE APLICACIONES PARA DISPOSITIVOS MOVILES (2977832)**

**SENA - SERVICIO NACIONAL DE APRENDIZAJE 2024**

## **INTRODUCCIÓN**

La programación orientada a objetos (POO) se fundamenta en la creación de modelos que representan el problema a resolver dentro de los programas. Este enfoque de programación reduce los errores y fomenta la reutilización del código. La POO, conocida también como OOP por sus siglas en inglés, es un paradigma que se basa en "objetos", los cuales contienen datos en forma de campos (atributos o propiedades) y métodos (código).

Los objetos pueden interactuar y modificar los valores de sus campos a través de sus métodos. Muchos lenguajes de programación modernos permiten la agrupación de objetos en bibliotecas, y también ofrecen la posibilidad de que los usuarios creen sus propias bibliotecas. Las características clave de la POO incluyen herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

## CONCEPTOS FUNDAMENTALES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

### DE PROCEDURAL A ORIENTADO A OBJETOS

Lenguajes de programación antiguos como C, Basic o COBOL seguían un estilo procedimental, donde el código consistía en una serie de instrucciones ejecutadas secuencialmente. Estos lenguajes utilizaban procedimientos (subrutinas o funciones) para encapsular funcionalidad, pero dependían de datos globales, dificultando el aislamiento de datos específicos. Este enfoque se centraba más en la lógica que en los datos.

En contraste, lenguajes modernos como C#, Java y otros utilizan paradigmas diferentes, siendo el más popular la Programación Orientada a Objetos (POO).

### PRINCIPALES CONCEPTOS DE LA POO

#### CLASES, OBJETOS E INSTANCIAS

El primer y más importante concepto de la POO es la distinción entre clase y objeto.

- **CLASE:** Es una plantilla que define de manera genérica cómo serán los objetos de un tipo específico. Por ejemplo, una clase Persona puede tener atributos como Nombre, Apellidos y Edad, y métodos como Hablar(), Caminar() y Comer().
- **OBJETO:** Es una instancia concreta de una clase. Al instanciar una clase, se crea un objeto real con existencia en memoria. Por ejemplo, un objeto de la clase Persona podría ser una entidad llamada Cristina López, de 37 años, capaz de hablar, caminar o comer.

```
public class Persona {  
    String nombre;  
    String apellidos;  
    int edad;  
  
    void hablar() { }  
    void caminar() { }  
    void comer() { }  
}
```

Para manejar eficientemente las clases y objetos generados con la POO, es necesario comprender cuatro principios fundamentales que ayudan a reducir la complejidad y evitar problemas.

## 1. ENCAPSULACIÓN

El concepto de encapsulamiento, aunque básico, es fundamental en la programación orientada a objetos (POO) y, a menudo, puede pasarse por alto debido a su simplicidad.

La encapsulación es una característica esencial en los lenguajes de POO que permite que toda la información relevante a un objeto permanezca dentro de él. Esto significa que los datos y detalles internos de un objeto están "encapsulados" y solo pueden ser accedidos mediante las propiedades y métodos definidos por la clase.

### DEFINICIÓN DE ENCAPSULACIÓN


- **ENCAPSULACIÓN:** La encapsulación implica que un objeto agrupa tanto sus características (datos) como su comportamiento (métodos) en una unidad indivisible.
- **OBJETO** = Características + Comportamiento
- **OBJETO** = Información + Proceso
- **OBJETO** = Atributos + Métodos


En resumen, el encapsulamiento se refiere a la consideración de un objeto como una unidad completa e indivisible que contiene tanto sus atributos como sus métodos, permitiendo así una gestión más eficiente y segura de los datos.


### EJEMPLO DE ENCAPSULACION(SEGÚN EL SOFTWARE A DESARROLLAR):


En una aplicación para una joyería artesanal utilizando **Java**, supongamos que queremos gestionar las piezas de joyería, como anillos, collares y pulseras. Vamos a crear una clase **Joya** con atributos privados y métodos públicos para acceder y modificar esos atributos, demostrando así la encapsulación. Cada pieza de joyería tiene un nombre, un material, un precio y un código único. Queremos asegurarnos de que estos datos estén protegidos para que no puedan ser modificados directamente de manera incorrecta.

### EXPLICACIÓN DEL CÓDIGO:

 **ATRIBUTOS PRIVADOS:** Los atributos nombre, material, precio y código están definidos como privados, lo que impide que se acceda directamente a ellos desde fuera de la clase.

 **MÉTODOS GETTER Y SETTER:** Estos métodos permiten obtener (getter) y modificar (setter) los valores de los atributos privados de forma controlada. Por ejemplo, en el método `setPrecio`, se asegura que el precio sea positivo antes de asignarlo.

 **ENCAPSULACIÓN:** La encapsulación se demuestra al proteger los datos internos de la clase **Joya**. Los métodos públicos `setPrecio`, `setMaterial`, etc., controlan cómo se pueden modificar los atributos, evitando así errores como establecer un precio negativo.

 **MÉTODO MAIN:** En el método `main`, se crea un objeto `collar` de la clase **Joya** y se muestran sus detalles. Luego se intenta cambiar el precio usando el método `setPrecio`, y el programa muestra los resultados, demostrando cómo la encapsulación protege los datos.

```

public class Joya {
    // Atributos privados
    private String nombre;
    private String material;
    private double precio;
    private String codigo;

    // Constructor
    public Joya(String nombre, String material, double precio, String codigo) {
        this.nombre = nombre;
        this.material = material;
        setPrecio(precio); // Uso del setter para la validación
        this.codigo = codigo;
    }

    // Método getter para el nombre
    public String getNombre() {
        return nombre;
    }

    // Método setter para el nombre
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```

```

// Método getter para el material
public String getMaterial() {
    return material;
}

// Método setter para el material
public void setMaterial(String material) {
    this.material = material;
}

// Método getter para el precio
public double getPrecio() {
    return precio;
}

// Método setter para el precio con validación
public void setPrecio(double precio) {
    if (precio > 0) {
        this.precio = precio;
    } else {
        System.out.println("El precio debe ser positivo.");
    }
}
}

```

```
// Método getter para el código
public String getCodigo() {
    return codigo;
}

// Método setter para el código
public void setCodigo(String codigo) {
    this.codigo = codigo;
}

// Método para mostrar los detalles de la joya
public void mostrarDetalles() {
    System.out.println("Nombre: " + nombre);
    System.out.println("Material: " + material);
    System.out.println("Precio: " + precio + " €");
    System.out.println("Código: " + codigo);
}
```

```
// Método principal para ejecutar el programa
public static void main(String[] args) {
    // Creación de una instancia de la clase Joya
    Joya collar = new Joya("Collar de Perlas", "Perlas", 150.0, "C001");

    // Mostrar detalles de la joya
    collar.mostrarDetalles();

    // Intentar modificar el precio usando el setter
    collar.setPrecio(200.0);

    // Mostrar detalles actualizados de la joya
    collar.mostrarDetalles();

    // Intentar establecer un precio negativo
    collar.setPrecio(-50.0); // No se aplicará por la validación
}
}
```

## 2. ABSTRACCIÓN

El principio de abstracción, como su nombre lo sugiere, implica que una clase debe representar las características visibles de una entidad, ocultando la complejidad interna asociada. Es decir, la abstracción nos permite utilizar una serie de atributos y comportamientos (propiedades y métodos) sin preocuparnos por los detalles internos de su implementación.

Una clase (y los objetos creados a partir de ella) debe exponer solo lo necesario para su uso. La forma en que se realizan las operaciones internamente no es relevante para los programas que utilizan estos objetos.


La abstracción está estrechamente relacionada con la encapsulación, pero va un paso más allá. No solo controla el acceso a la información, sino que también oculta la complejidad de los procesos implementados, permitiendo a los usuarios interactuar con objetos de manera sencilla y eficiente, sin necesidad de conocer los detalles internos.


### EJEMPLO DE ABSTRACCIÓN ( SEGÚN EL SOFTWARE A DESARROLLAR):


Supongamos que estamos desarrollando una aplicación para una joyería artesanal. En esta joyería, vendemos diferentes tipos de joyas como anillos, collares y pulseras. Cada tipo de joya tiene características comunes, como un nombre, un material y un precio, pero también puede tener características específicas.


En este caso, utilizaremos la abstracción para definir una clase abstracta Joya, que servirá como base para diferentes tipos de joyas. Esta clase abstracta contendrá los atributos y métodos comunes, mientras que las clases derivadas (como Anillo, Collar y Pulsera) implementarán los detalles específicos.

### EXPLICACIÓN DEL CÓDIGO

 **CLASE ABSTRACTA JOYA:** La clase Joya es abstracta, lo que significa que no se puede instanciar directamente. Define los atributos comunes (nombre, material, precio) y un método abstracto mostrarDetalles(), que debe ser implementado por cualquier clase derivada.

 **CLASES CONCRETAS ANILLO Y COLLAR:** Las clases Anillo y Collar heredan de Joya y proporcionan su propia implementación del método mostrarDetalles(). Cada una también define atributos específicos, como talla para los anillos y longitud para los collares.

 **ABSTRACCIÓN EN ACCIÓN:** Al definir la clase Joya como abstracta, estamos enfocándonos en lo que es común entre todas las joyas y delegando los detalles específicos a las clases concretas que heredan de ella. Esto permite manejar diferentes tipos de joyas de manera uniforme y facilita la expansión futura (por ejemplo, añadir nuevas clases como Pulsera).

 **MÉTODO MAIN:** En el método main, se crean instancias de Anillo y Collar, y se muestran sus detalles utilizando el método mostrarDetalles(). Este método abstraer la lógica de mostrar la información, asegurando que cada tipo de joya se maneje correctamente sin necesidad de saber sus detalles internos.

```
// Clase abstracta Joya
public abstract class Joya {
    private String nombre;
    private String material;
    private double precio;

    public Joya(String nombre, String material, double precio) {
        this.nombre = nombre;
        this.material = material;
        this.precio = precio;
    }

    // Método abstracto para mostrar detalles
    public abstract void mostrarDetalles();

    // Getters para los atributos comunes
    public String getNombre() {
        return nombre;
    }

    public String getMaterial() {
        return material;
    }
}
```

```
public double getPrecio() {
    return precio;
}

// Clase concreta Anillo que hereda de Joya
class Anillo extends Joya {
    private String talla;

    public Anillo(String nombre, String material, double precio, String talla) {
        super(nombre, material, precio);
        this.talla = talla;
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Nombre: " + getNombre());
        System.out.println("Material: " + getMaterial());
        System.out.println("Precio: " + getPrecio() + " €");
        System.out.println("Talla: " + talla);
    }
}
```



```
// Clase concreta Collar que hereda de Joya
class Collar extends Joya {
    private double longitud;

    public Collar(String nombre, String material, double precio, double longitud) {
        super(nombre, material, precio);
        this.longitud = longitud;
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Nombre: " + getNombre());
        System.out.println("Material: " + getMaterial());
        System.out.println("Precio: " + getPrecio() + " €");
        System.out.println("Longitud: " + longitud + " cm");
    }
}
```

```
// Clase principal para ejecutar el programa
public class Joyería {
    public static void main(String[] args) {
        // Creación de objetos de las clases Anillo y Collar
        Anillo anillo = new Anillo("Anillo de Oro", "Oro", 500.0, "16");
        Collar collar = new Collar("Collar de Plata", "Plata", 300.0, 45.0);

        // Mostrar detalles de las joyas
        anillo.mostrarDetalles();
        System.out.println();
        collar.mostrarDetalles();
    }
}
```

### 3. HERENCIA

En genética, se dice que una persona hereda ciertos rasgos de sus padres, como el color de los ojos o la piel, o incluso ciertas enfermedades genéticas. De manera similar, en la programación orientada a objetos (POO), cuando una clase hereda de otra, adquiere todos los atributos y métodos de la clase original.

Una clase actúa como un molde que define cómo es y cómo se comporta una entidad. Cuando una clase hereda de otra, adopta todos los atributos y comportamientos de la primera, pero también puede agregar nuevos y modificar los existentes. La clase de la que se hereda se llama clase base, mientras que la clase que hereda se llama clase derivada.

Por ejemplo, en un juego con personajes, podemos crear clases especializadas a partir de una clase general llamada


La herencia es una de las características más poderosas de la POO porque promueve la reutilización del código y permite la especialización y personalización de clases de manera eficiente.


#### EJEMPLO DE HERENCIA ( SEGÚN EL SOFTWARE A DESARROLLAR):

En una joyería artesanal, es común tener diferentes tipos de joyas como anillos, collares, pulseras, etc. Todas estas joyas comparten algunas características comunes, como el nombre, el material y el precio. Sin embargo, cada tipo de joya puede tener características específicas adicionales.

Podemos utilizar la herencia para definir una clase base Joya, que contendrá las características comunes, y luego crear clases derivadas como Anillo, Collar y Pulsera, que heredarán de Joya y podrán agregar sus propias características.


#### EXPLICACIÓN DEL CÓDIGO:


 **CLASE BASE JOYA:** Esta clase contiene los atributos comunes a todas las joyas, como nombre, material y precio. También incluye un método mostrarDetalles() que imprime los detalles de la joya.

 **CLASES DERIVADAS ANILLO Y COLLAR:** Estas clases heredan de Joya utilizando la palabra clave extends.

- **Anillo:** Además de los atributos heredados de Joya, tiene un atributo adicional talla y su propio constructor. Sobrescribe el método mostrarDetalles() para mostrar también la talla del anillo.

- **Collar:** Además de los atributos heredados de Joya, tiene un atributo adicional longitud y su propio constructor. También sobrescribe el método mostrarDetalles() para mostrar la longitud del collar.

 **USO DE LA HERENCIA:** La herencia permite que Anillo y Collar reutilicen el código de la clase Joya y solo añadan o sobrescriban lo que es específico para cada tipo de joya.

 **MÉTODO MAIN:** En el método main, se crean instancias de Anillo y Collar, y se muestran sus detalles utilizando el método mostrarDetalles(). Este método aprovecha la herencia para reutilizar el código de la clase Joya y extenderlo con detalles específicos.

```
// Clase base Joya
public class Joya {
    private String nombre;
    private String material;
    private double precio;

    // Constructor de la clase Joya
    public Joya(String nombre, String material, double precio) {
        this.nombre = nombre;
        this.material = material;
        this.precio = precio;
    }

    // Métodos getters
    public String getNombre() {
        return nombre;
    }

    public String getMaterial() {
        return material;
    }
}
```

```
public double getPrecio() {
    return precio;
}

// Método para mostrar detalles de la joya
public void mostrarDetalles() {
    System.out.println("Nombre: " + nombre);
    System.out.println("Material: " + material);
    System.out.println("Precio: " + precio + " €");
}

// Clase derivada Anillo que hereda de Joya
class Anillo extends Joya {
    private String talla;

    // Constructor de la clase Anillo
    public Anillo(String nombre, String material, double precio, String talla) {
        super(nombre, material, precio); // Llama al constructor de la superclase Joya
        this.talla = talla;
    }
}
```

```

// Método para mostrar detalles del anillo
@Override
public void mostrarDetalles() {
    super.mostrarDetalles(); // Llama al método de la superclase
    System.out.println("Talla: " + talla);
}
}

// Clase derivada Collar que hereda de Joya
class Collar extends Joya {
    private double longitud;

    // Constructor de la clase Collar
    public Collar(String nombre, String material, double precio, double longitud) {
        super(nombre, material, precio); // Llama al constructor de la superclase
        this.longitud = longitud;
    }
}

```

```

// Método para mostrar detalles del collar
@Override
public void mostrarDetalles() {
    super.mostrarDetalles(); // Llama al método de la superclase
    System.out.println("Longitud: " + longitud + " cm");
}
}

// Clase principal para ejecutar el programa
public class Joyería {
    public static void main(String[] args) {
        // Creación de objetos de las clases Anillo y Collar
        Anillo anillo = new Anillo("Anillo de Oro", "Oro", 500.0, "16");
        Collar collar = new Collar("Collar de Plata", "Plata", 300.0, 45.0);

        // Mostrar detalles de las joyas
        System.out.println("Detalles del Anillo:");
        anillo.mostrarDetalles();

        System.out.println("\nDetalles del Collar:");
        collar.mostrarDetalles();
    }
}

```

#### 4. POLIMORFISMO

El término polimorfismo proviene del griego "polys" (muchos) y "morfo" (forma), y significa "capacidad de tomar muchas formas". En programación orientada a objetos (POO), polimorfismo se refiere a la habilidad de usar objetos de diferentes clases, con una base común, de manera intercambiable, sin necesidad de conocer su clase específica.

El polimorfismo puede ser más complejo, ya que también incluye la sobrecarga de métodos y el uso de interfaces. Sin embargo, el concepto básico es que permite utilizar objetos de manera generalizada, mientras que internamente se comportan según su tipo específico.

##### EJEMPLO DE POLIMORFISMO ( SEGÚN EL SOFTWARE A DESARROLLAR):

Supongamos que estamos desarrollando una aplicación para una joyería artesanal que gestiona diferentes tipos de joyas: anillos, collares y pulseras. Cada uno de estos tipos de joyas tiene un método `mostrarDetalles()` que muestra información específica sobre la joya.

Utilizaremos polimorfismo para crear una lista de joyas de diferentes tipos (anillos, collares, pulseras) y recorrer esa lista mostrando los detalles de cada joya, sin necesidad de saber de antemano de qué tipo específico es cada joya.

##### EXPLICACIÓN DEL CÓDIGO:

🚦 **CLASE BASE JOYA:** Esta clase contiene atributos comunes (nombre, material, precio) y un método `mostrarDetalles()` que muestra la información básica de una joya.

🚦 **CLASES DERIVADAS ANILLO, COLLAR Y PULSERA:** Estas clases heredan de Joya y sobrescriben el método `mostrarDetalles()` para agregar detalles específicos (talla en Anillo, longitud en Collar, tipo de cierre en Pulsera).

🚦 **POLIMORFISMO:** En el método `main`, creamos un array de tipo Joya, pero lo llenamos con objetos de diferentes clases (Anillo, Collar, Pulsera). Luego, iteramos sobre el array y llamamos al método `mostrarDetalles()` en cada objeto. Gracias al polimorfismo, Java invoca el método correspondiente a la clase específica del objeto, incluso si se maneja a través de una referencia de tipo Joya.

🚦 **BENEFICIOS DEL POLIMORFISMO:** Este enfoque permite manejar diferentes tipos de objetos de una manera uniforme, lo que facilita la extensión y el mantenimiento del código. Por ejemplo, si se agrega una nueva clase derivada (como Pendiente), el código existente no necesita cambios para manejar los nuevos objetos.

```
// Clase base Joya
public class Joya {
    private String nombre;
    private String material;
    private double precio;

    // Constructor de la clase Joya
    public Joya(String nombre, String material, double precio) {
        this.nombre = nombre;
        this.material = material;
        this.precio = precio;
    }

    // Método mostrarDetalles para sobrescribir en las clases derivadas
    public void mostrarDetalles() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Material: " + material);
        System.out.println("Precio: " + precio + " €");
    }
}
```

```
// Clase derivada Anillo que hereda de Joya
class Anillo extends Joya {
    private String talla;

    public Anillo(String nombre, String material, double precio, String talla) {
        super(nombre, material, precio);
        this.talla = talla;
    }

    @Override
    public void mostrarDetalles() {
        super.mostrarDetalles();
        System.out.println("Talla: " + talla);
    }
}

// Clase derivada Collar que hereda de Joya
class Collar extends Joya {
    private double longitud;

    public Collar(String nombre, String material, double precio, double longitud) {
        super(nombre, material, precio);
        this.longitud = longitud;
    }
}
```

```

@Override
public void mostrarDetalles() {
    super.mostrarDetalles();
    System.out.println("Longitud: " + longitud + " cm");
}

// Clase derivada Pulsera que hereda de Joya
class Pulsera extends Joya {
    private String tipoCierre;

    public Pulsera(String nombre, String material, double precio, String tipoCierre) {
        super(nombre, material, precio);
        this.tipoCierre = tipoCierre;
    }

    @Override
    public void mostrarDetalles() {
        super.mostrarDetalles();
        System.out.println("Tipo de Cierre: " + tipoCierre);
    }
}

```

```

// Clase principal para ejecutar el programa
public class Joyería {

    public static void main(String[] args) {

        // Creación de diferentes objetos de joyas
        Joya anillo = new Anillo("Anillo de Oro", "Oro", 500.0, "16");
        Joya collar = new Collar("Collar de Plata", "Plata", 300.0, 45.0);
        Joya pulsera = new Pulsera("Pulsera de Cuero", "Cuero", 100.0, "Broche");

        // Array de joyas de tipo Joya (polimorfismo)
        Joya[] joyas = {anillo, collar, pulsera};

        // Recorrer el array y mostrar detalles de cada joya (polimorfismo en acción)
        for (Joya joya : joyas) {
            System.out.println("\nDetalles de la Joya:");
            joya.mostrarDetalles(); // Polimorfismo: invoca el método correspondiente
        }
    }
}

```

