




Trabajo Práctico 4: Programación Orientada a Objetos II

Alumno: Calcatelli Renzo - rcalcatelli@gmail.com

Comisión: M2025-1

Materia: Programación II

Profesor: Ariel Enferrel

 [Enlace](#) al repositorio de GitHub

Trabajo Práctico 4: Programación Orientada a Objetos II

Introducción

En este trabajo práctico, el objetivo es aplicar varios conceptos clave de la Programación Orientada a Objetos (POO) en Java para modelar la clase **Empleado**. La implementación se centra en el uso de **this**, la sobrecarga de constructores y métodos, el encapsulamiento y los miembros estáticos. La idea es que el código sea claro, modular y fácil de mantener.

A continuación, se presenta la implementación de la clase **Empleado** y de una clase de prueba (**TestEmpleados**) para demostrar su correcto funcionamiento.

Explicación Detallada de la Implementación - Conceptos de POO

1. Uso de **this**

¿Qué es **this**?

La palabra clave **this** es una **referencia implícita** al objeto actual (la instancia) que está siendo utilizada en ese momento. Es como un "pronombre" que se refiere al objeto que está ejecutando el método o constructor.

¿Cuándo y por qué se usa **this**?

Caso 1: Ambigüedad entre parámetros y atributos

```
public Empleado(int id, String nombre, String puesto, double salario) {  
    this.id = id;           // this.id se refiere al atributo de la clase  
    this.nombre = nombre; // nombre (sin this) se refiere al parámetro  
    this.puesto = puesto;  
    this.salario = salario;  
}
```

Sin **this** (INCORRECTO):

```
public Empleado(int id, String nombre, String puesto, double salario) {  
    id = id;           // Se asignando el parámetro a sí mismo  
    nombre = nombre; // No se asigna al atributo de la clase  
}
```

Caso 2: Llamada a otro constructor

```
public Empleado(String nombre, String puesto) {  
    this(contadorId, nombre, puesto, 30000.0); // Llama al constructor  
    principal  
}
```

Ventajas del uso de **this**:

- **Claridad:** Hace explícito que se refiere al atributo de la instancia.
 - **Evita errores:** Previene asignaciones incorrectas.
 - **Legibilidad:** El código es más fácil de entender.
 - **Buenas prácticas:** Es una convención estándar en Java.
-

2. Constructores Sobrecargados

¿Qué es la sobrecarga de constructores?

La **sobrecarga de constructores** permite tener múltiples constructores en la misma clase, cada uno con **diferentes parámetros**. Esto proporciona flexibilidad al crear objetos.

Reglas para la sobrecarga:

1. **Misma clase:** Todos los constructores pertenecen a la misma clase.
2. **Mismo nombre:** Todos tienen el nombre de la clase.
3. **Diferentes parámetros:** Deben diferir en número, tipo o orden de parámetros.
4. **Diferente funcionalidad:** Cada constructor puede tener una lógica diferente.

Ejemplo en el código:

Constructor Completo:

```
public Empleado(int id, String nombre, String puesto, double salario) {  
    this.id = id;  
    this.nombre = nombre;  
    this.puesto = puesto;  
    this.salario = salario;  
    totalEmpleados++;  
}
```

Uso: Cuando se tienen todos los datos del empleado desde el principio.

Constructor Simplificado:

```
public Empleado(String nombre, String puesto) {  
    this(contadorId, nombre, puesto, 30000.0); // Delegación de constructor  
}
```

Uso: Cuando solo se tienen datos básicos y se quieren obtener valores automáticos.

Ventajas de los constructores sobrecargados:

- **Flexibilidad:** Diferentes formas de crear objetos según el contexto.
- **Reutilización:** Un constructor puede llamar a otro (delegación).
- **Valores por defecto:** Se pueden proporcionar valores predeterminados.
- **API más rica:** Al sobrecargar constructores y métodos, tu clase se vuelve una herramienta más intuitiva y flexible.

Patrón de Delegación:

```
// Constructor más simple llama al más complejo  
public Empleado(String nombre, String puesto) {  
    this(contadorId, nombre, puesto, 30000.0);  
    // ↑ Delegación: este constructor hace el “trabajo pesado”  
}
```

3. Métodos Sobrecargados (Overloading)

¿Qué es la sobrecarga de métodos?

La sobrecarga de métodos permite definir múltiples métodos con el mismo nombre pero diferentes parámetros en la misma clase.

Reglas para la sobrecarga de métodos:

1. **Mismo nombre:** Todos los métodos tienen idéntico nombre.
2. **Diferentes parámetros:** Deben diferir en:
 - Número de parámetros.
 - Tipo de parámetros.
 - Orden de parámetros.
3. **Puede diferir el tipo de retorno:** Pero no es suficiente por sí solo.

Ejemplo en el código:

Versión con porcentaje:

```
public void actualizarSalario(double porcentajeAumento) {  
    double aumento = this.salario * (porcentajeAumento / 100);  
    this.salario += aumento;  
    System.out.println("Salario actualizado con " + porcentajeAumento + "%  
de aumento.");  
}
```

Versión con cantidad fija:

```
public void actualizarSalario(int cantidadFija) {  
    this.salario += cantidadFija;  
    System.out.println("Salario actualizado con aumento fijo de $" +  
cantidadFija);  
}
```

¿Cómo decide Java qué método usar?

Java utiliza la resolución de sobrecarga basada en:

1. Número de argumentos.
2. Tipos exactos de argumentos.
3. Promoción de tipos (si es necesario).

```
empleado.actualizarSalario(15.0); // Llama a la versión double  
(porcentaje)  
empleado.actualizarSalario(5000); // Llama a la versión int (cantidad  
fija)
```

Ventajas de la sobrecarga de métodos:

- **API intuitiva:** Un solo nombre para operaciones relacionadas.
 - **Flexibilidad:** Diferentes formas de realizar la misma operación conceptual.
 - **Legibilidad:** Código más natural y fácil de recordar.
 - **Polimorfismo estático:** Decidido en tiempo de compilación.
-

4. Método toString()

¿Qué es toString()?

toString() es un método **heredado de la clase Object** que devuelve una representación en cadena del objeto. Por defecto, retorna el nombre de la clase + código hash del objeto.

Sin override (comportamiento por defecto):

```
Empleado emp = new Empleado("Juan", "Desarrollador");
System.out.println(emp); // Imprime: Empleado@1b6d3586 (no muy útil)
```

Con override (el código):

```
@Override
public String toString() {
    return String.format("=== EMPLEADO ===\n" +
        "ID: %d\n" +
        "Nombre: %s\n" +
        "Puesto: %s\n" +
        "Salario: $%.2f\n" +
        "=====",
        this.id, this.nombre, this.puesto, this.salario);
}
```

Características importantes:

- **@Override**: Anotación que indica que estás reemplazando un método heredado.
- **String.format()**: Permite formateo profesional de la cadena.
- **Información útil**: Muestra todos los datos relevantes del objeto.
- **Formato legible**: Estructurado para fácil lectura.

¿Cuándo se llama automáticamente toString()?

```
System.out.println(empleado);           // Llamada implícita
System.out.println("Empleado: " + empleado); // Llamada implícita en concatenación
empleado.toString();                     // Llamada explícita
```

Ventajas de implementar toString():

- **Depuración:** Facilita encontrar errores mostrando el estado del objeto.
 - **Logging:** Útil para registros de aplicación.
 - **Interfaz de usuario:** Para mostrar información al usuario.
 - **Testing:** Comparar estados de objetos en pruebas.
-

5. Atributos y Métodos Estáticos

¿Qué significa "estático"?

Un miembro estático pertenece a la clase en lugar de a una instancia específica. Existe una sola copia que es compartida por todas las instancias.

Atributos Estáticos:

```
private static int totalEmpleados = 0; // Una sola variable para toda la
clase
private static int contadorId = 1;    // Compartida por todos los objetos
```

Características de los atributos estáticos:

- **Una sola copia:** No importa cuántos objetos crees, solo hay una variable.
- **Compartido:** Todas las instancias ven el mismo valor.
- **Inicialización:** Se inicializa cuando la clase se carga por primera vez.
- **Persistencia:** Mantiene su valor durante toda la ejecución del programa.

Métodos Estáticos:

```
public static int mostrarTotalEmpleados() {
    return totalEmpleados; // Solo puede acceder a miembros estáticos
}
```

Características de los métodos estáticos:

- **Sin instancia:** Se puede llamar sin crear un objeto.
- **Acceso limitado:** Solo puede acceder a miembros estáticos.
- **Sintaxis de llamada:** **ClaseNombre.metodoEstatico()**.
- **Utilidad:** Operaciones que no dependen de una instancia específica.

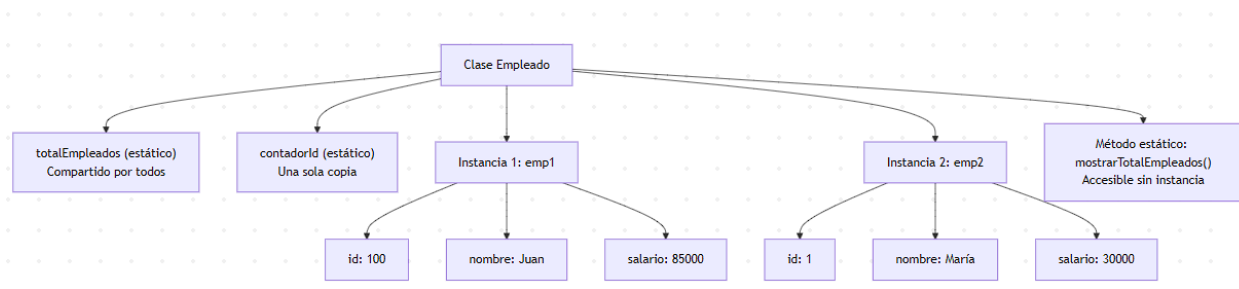
Ejemplo de uso:

```
// Sin crear ningún objeto Empleado
int total = Empleado.mostrarTotalEmpleados();
Empleado.mostrarEstadisticas();
```

¿Por qué usar miembros estáticos?

- **Contadores globales:** Como **totalEmpleados**.
- **Constantes:** Valores que no cambian (**final static**).
- **Métodos utilitarios:** Funciones que no necesitan estado de instancia.
- **Patrón Singleton:** Para limitar a una sola instancia.

Diagrama conceptual:



6. Encapsulamiento

¿Qué es el encapsulamiento?

El encapsulamiento es el principio de ocultar los detalles internos de una clase y controlar el acceso a sus datos a través de métodos públicos.

Niveles de acceso en Java:

- **private**: Solo accesible dentro de la misma clase.
- **protected**: Accesible en la misma clase, subclases y mismo paquete.
- **public**: Accesible desde cualquier lugar.
- **(default)**: Accesible en el mismo paquete.

En el código:

```
// Atributos privados (encapsulados)
private int id;
private String nombre;
private String puesto;
private double salario;

// Métodos públicos para acceso controlado
public int getId() { return this.id; }
public void setId(int id) { this.id = id; }
```

Getters (Métodos de acceso):

```
public String getNombre() {
    return this.nombre; // Retorna una copia del valor
}
```

Función: Proporcionar acceso de **solo lectura** a los atributos privados.

Setters (Métodos modificadores):

```
public void setSalario(double salario) {
    if (salario >= 0) { // Validación
        this.salario = salario;
    } else {
        System.out.println("Error: El salario no puede ser negativo");
    }
}
```

Función: Proporcionar acceso de **escritura controlada** con validación.

Ventajas del encapsulamiento:

1. **Protección de datos:** Evita modificaciones no autorizadas.
2. **Validación:** Se pueden verificar datos antes de asignarlos.
3. **Flexibilidad:** Cambiar implementación interna sin afectar el código cliente.
4. **Debugging:** Controlar dónde y cómo se modifican los datos.
5. **Mantenimiento:** Fácil modificar la lógica de acceso a datos.

Ejemplo de protección:

```
// Sin encapsulamiento (MALO)
empleado.salario = -5000; // Salario negativo

// Con encapsulamiento (BUENO)
empleado.setSalario(-5000); // Validación previene el error
```

Principio de ocultamiento de información:

- **Interfaz pública:** Lo que los usuarios de la clase necesitan saber.
 - **Implementación privada:** Los detalles internos que pueden cambiar.
-

7. Conceptos Adicionales Aplicados

- **Inmutabilidad parcial:** Los getters retornan copias o referencias a objetos inmutables (como String), evitando modificaciones accidentales.
- **Validación de datos:** Los setters incluyen lógica de validación para mantener la integridad de los datos.
- **Separación de responsabilidades:** Cada método tiene una responsabilidad específica y bien definida.

Convenciones de nomenclatura:

- **CamelCase** para métodos y variables.
 - **PascalCase** para clases.
 - **Nombres descriptivos y significativos.**
-

Beneficios de la Programación Orientada a Objetos

Estos conceptos aplicados en el trabajo no son solo teoría; son la base para escribir código de calidad profesional. Al usarlos, este proyecto, en este caso, gana en varios aspectos clave:

1. **Mantenibilidad:** El código es mucho más fácil de modificar y extender. Por ejemplo, si mañana necesitamos agregar un nuevo método a la clase **Empleado** (como una forma de calcular bonos), podemos hacerlo sin tener que tocar el código principal del programa. Todo está organizado dentro de la clase.
2. **Reutilización:** Una clase como **Empleado** se convierte en un componente que podemos usar en cualquier otro proyecto que requiera gestionar personal. No tenemos que escribir el mismo código una y otra vez; simplemente importamos la clase y ya está lista para usar.

3. **Robustez:** El encapsulamiento es clave. Al restringir el acceso directo a los atributos, evitamos que otro programador cambie el salario de un empleado de forma incorrecta o accidental. Solo se puede hacer a través de los métodos diseñados para esa tarea, lo que minimiza los errores.
4. **Legibilidad:** El código se "autodocumenta". Cuando vemos `emp2.actualizarSalario(5000)`, sabemos exactamente qué está pasando, sin tener que revisar el código interno del método. Además, la sobrecarga nos permite usar nombres de métodos claros y descriptivos, como el mismo `actualizarSalario()` para ambos tipos de aumentos.
5. **Escalabilidad:** El diseño modular que logramos soporta el crecimiento. Si la empresa tiene 10 empleados o 10,000, la clase **Empleado** funcionará igual de bien. La lógica está aislada y no colapsa a medida que el sistema se hace más grande y complejo.
6. **Testing:** Al tener cada parte de la lógica bien encapsulada en la clase, se facilita la creación de pruebas unitarias. Por ejemplo, podemos probar el método `actualizarSalario()` de forma aislada para asegurarnos de que el cálculo siempre es correcto, sin que nos afecte el resto del programa.

Estos conceptos no son solo para pasar el trabajo; son las herramientas que nos permiten desarrollar software bien diseñado, que puede crecer y adaptarse a futuros cambios de manera eficiente. Son los pilares para un desarrollo de software de calidad.