

Trabajo Práctico N° 4

Programación orientada a objetos

Comision 16

Marinoni Macarena <marinonimacarena@gmail.com>

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional

Programación II

Profesor: Cortez, Alberto

Tutor: Bianchi, Neyén

26/09/2025

ÍNDICE

Objetivo general	3
Marco teórico	4
Caso práctico	5
Conclusión	8
Anexo	9

Objetivo general

Comprender y aplicar conceptos de Programación Orientada a Objetos en Java mediante la construcción de clases simples y su verificación desde un main. Para ello, aplicará correctamente el uso de this en constructores y métodos, la sobrecarga de constructores y de métodos, el encapsulamiento mediante atributos privados con getters/setters, y el empleo de miembros estáticos y de toString() para describir el estado de los objetos. El objetivo es lograr código modular, legible y fácil de mantener, comprendiendo la diferencia entre miembros de instancia y miembros de clase y reforzando buenas prácticas de validación de datos.

Marco teórico

La Programación Orientada a Objetos propone modelar el programa como un conjunto de clases (planos) y objetos (instancias) con estado y comportamiento. En este trabajo se enfoca en cinco ejes prácticos:

1) this en Java.

this refiere explícitamente a la instancia actual. Se utiliza para distinguir parámetros de atributos y para encadenar constructores con this(...), evitando duplicación de código y asegurando una inicialización coherente del estado del objeto.

2) Constructores y sobrecarga.

Un constructor establece el estado inicial del objeto. La sobrecarga permite definir múltiples formas de instanciación para dar flexibilidad: por ejemplo, un constructor “completo” y otro “mínimo” con valores por defecto. Esta técnica mejora la usabilidad de la clase y reduce errores de inicialización.

3) Métodos sobrecargados.

La sobrecarga de métodos habilita varias versiones de un mismo nombre con firmas diferentes (parámetros distintos). Se emplea para ofrecer alternativas de operación —por ejemplo, actualizar un salario por porcentaje o por monto fijo— sin cambiar el nombre del método, favoreciendo una interfaz pública coherente.

4) Encapsulamiento (getters/setters).

El encapsulamiento restringe el acceso directo a los atributos marcándolos como private y exponiendo métodos de acceso controlado. Con getters/setters, la clase puede validar datos y mantener sus invariantes de negocio, reduciendo errores y acoplamiento entre componentes.

5) Miembros estáticos y toString().

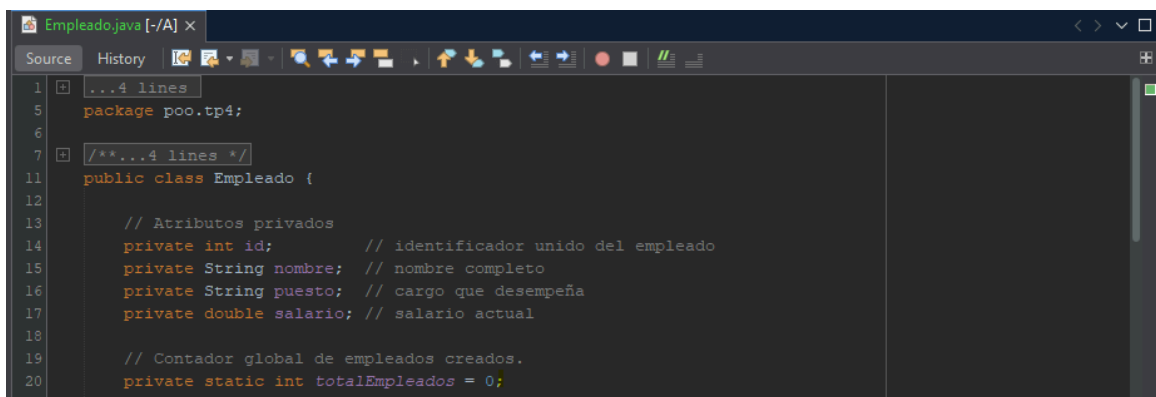
Los atributos estáticos pertenecen a la clase y son compartidos por todas las instancias (por ejemplo, un contador global de objetos creados). Los métodos estáticos se invocan sin instanciar. Por su parte, sobrescribir toString() brinda una representación legible del estado del objeto, útil para impresión y depuración, evitando el formato por defecto poco informativo. En conjunto, estos recursos ayudan a diferenciar responsabilidades de clase vs instancia y a inspeccionar el estado de forma clara.

Caso práctico

En el desarrollo de este trabajo se modela una clase Empleado que represente a un trabajador en una empresa. Esta clase incluye constructores sobrecargados, métodos sobrecargados y el uso de atributos aplicando encapsulamiento y métodos estáticos para llevar control de los objetos creados.

Se definen para la clase Empleado los siguientes atributos:

- int id: Identificador único del empleado.
- String nombre: Nombre completo.
- String puesto: Cargo que desempeña.
- double salario: Salario actual.
- static int totalEmpleados: Contador global de empleados creados.

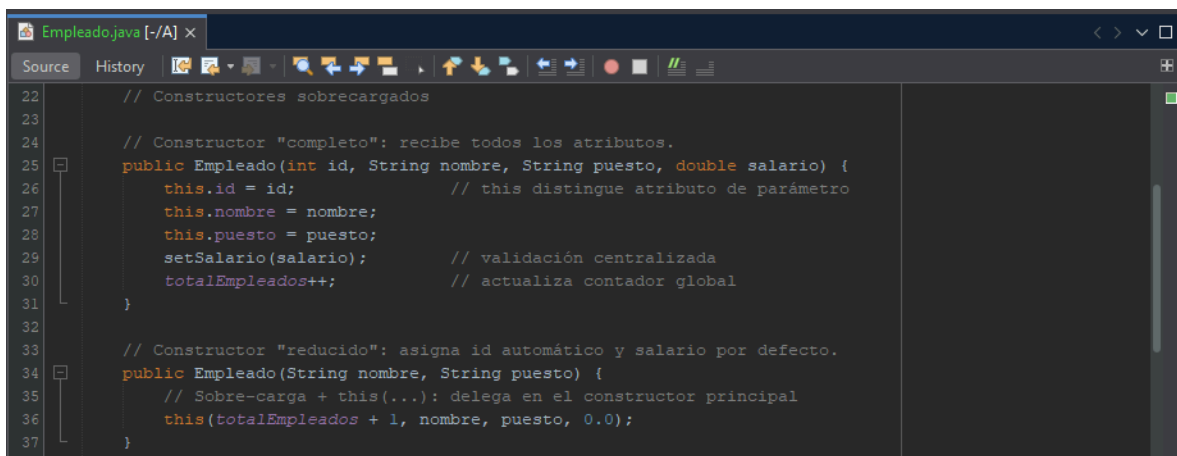


```

1  ...4 lines
5  package poo.tp4;
6
7  /**...4 lines */
11 public class Empleado {
12
13     // Atributos privados
14     private int id; // identificador unico del empleado
15     private String nombre; // nombre completo
16     private String puesto; // cargo que desempeña
17     private double salario; // salario actual
18
19     // Contador global de empleados creados.
20     private static int totalEmpleados = 0;
  
```

Luego se definen constructores sobrecargados con diferentes objetivos:

- Uno que recibe todos los atributos como parámetros.
- Uno que recibe solo nombre y puesto, asignando un id automático y un salario por defecto.



```

22 // Constructores sobrecargados
23
24 // Constructor "completo": recibe todos los atributos.
25 public Empleado(int id, String nombre, String puesto, double salario) {
26     this.id = id; // this distingue atributo de parametro
27     this.nombre = nombre;
28     this.puesto = puesto;
29     setSalario(salario); // validación centralizada
30     totalEmpleados++; // actualiza contador global
31 }
32
33 // Constructor "reducido": asigna id automático y salario por defecto.
34 public Empleado(String nombre, String puesto) {
35     // Sobre-carga + this(...): delega en el constructor principal
36     this(totalEmpleados + 1, nombre, puesto, 0.0);
37 }
  
```

También se desarrollan métodos sobrecargados para actualizar el salario del empleado de dos formas diferentes:

- Uno que reciba un porcentaje de aumento.
- Otro que reciba una cantidad fija a aumentar.

```

39 // Métodos sobrecargados actualizarSalario()
40
41 // Actualiza con % de aumento
42 public void actualizarSalario(double porcentaje) {
43     double delta = salario * (porcentaje / 100.0);
44     this.salario += delta;
45 }
46
47 // Actualiza con valor fijo
48 public void actualizarSalario(double montoFijo, boolean esMontoFijo) {
49     if (!esMontoFijo) return;
50     this.salario += montoFijo;
51 }

```

Se aplica el método `toString()` para mostrar id, nombre, puesto y salario de forma legible y también el método estático `mostrarTotalEmpleados()` para retornar el total de empleados creados hasta el momento.

```

52
53 // Metodo toString()
54 // Mostrar id, nombre, puesto y salario de forma legible.
55 @Override
56 public String toString() {
57     return String.format("Empleado{id=%d, nombre='%s', puesto='%s', salario=%.2f}",
58         id, nombre, puesto, salario);
59 }
60
61 // Metodo mostrarTotalEmpleados
62 // Retornar el total de empleados creados hasta el momento.
63
64 public static int mostrarTotalEmpleados() {
65     return totalEmpleados;
66 }

```

Por último, se generan encapsulamiento en los atributos con la finalidad de restringir el acceso directo a los atributos de la clase:

```

67
68 // Encapsulamiento en los atributos
69 // --- Getters/Setters ---
70
71 public int getId() { return id; }
72 public void setId(int id) { if (id > 0) this.id = id; }
73
74 public String getNombre() { return nombre; }
75 public void setNombre(String nombre) {
76     if (nombre != null && !nombre.trim().isEmpty()) this.nombre = nombre.trim();
77 }
78
79 public String getPuesto() { return puesto; }
80 public void setPuesto(String puesto) {
81     if (puesto != null && !puesto.trim().isEmpty()) this.puesto = puesto.trim();
82 }
83
84 public double getSalario() { return salario; }
85 public void setSalario(double salario) { this.salario = Math.max(0.0, salario); }
86
87 }

```

Para verificar la correcta implementación de la clase Empleado con todos los puntos anteriores se crea una clase de prueba con método main que:

- Instancie varios objetos usando ambos constructores.
- Aplique los métodos actualizarSalario() sobre distintos empleados.
- Imprima la información de cada empleado con toString().
- Muestre el total de empleados creados con mostrarTotalEmpleados().

```

5 package poo.tp4;
6 /**...4 lines */
10 public class TP_Unidad4 {
11     /**...3 lines */
14     public static void main(String[] args) {
15         // Constructor reducido (salario arranca en 0.0)
16         Empleado e1 = new Empleado("Joaquin Gaete", "Analista Tecnico");
17         Empleado e2 = new Empleado("Ma. Guadalupe De Angelis", "Analista de Suscripción");
18
19         // Setear salario inicial explicitamente
20         e1.setSalario(180_000.0);
21         e2.setSalario(200_000.0);
22
23         // Constructor completo
24         Empleado e3 = new Empleado(100, "Macarena Marinoni", "Supervisora Tecnica", 250_000);
25
26         System.out.println("Estado inicial");
27         System.out.println(e1);
28         System.out.println(e2);
29         System.out.println(e3);
30
31         // Aumentos
32         e2.actualizarSalario(12.5); // +12.5% a e2
33         e3.actualizarSalario(50_000.0, true); // +$50.000 a e3
34
35         System.out.println("\nLuego de actualizar salario");
36         System.out.println(e1);
37         System.out.println(e2);
38         System.out.println(e3);
39
40         System.out.println("\nTotal empleados: " + Empleado.mostrarTotalEmpleados());
    
```

Output:

```

run:
Estado inicial
Empleado{id=1, nombre='Joaquin Gaete', puesto='Analista Tecnico', salario=180000,00}
Empleado{id=2, nombre='Ma. Guadalupe De Angelis', puesto='Analista de Suscripción', salario=200000,00}
Empleado{id=100, nombre='Macarena Marinoni', puesto='Supervisora Tecnica', salario=250000,00}

Luego de actualizar salario
Empleado{id=1, nombre='Joaquin Gaete', puesto='Analista Tecnico', salario=180000,00}
Empleado{id=2, nombre='Ma. Guadalupe De Angelis', puesto='Analista de Suscripción', salario=225000,00}
Empleado{id=100, nombre='Macarena Marinoni', puesto='Supervisora Tecnica', salario=300000,00}

Total empleados: 3
BUILD SUCCESSFUL (total time: 0 seconds)
    
```

Conclusión

El trabajo presentado evidencia que se aplicó de forma correcta y ordenada los fundamentos de POO en Java para resolver el caso Empleado. Se modeló una clase con encapsulamiento (atributos privados y acceso controlado mediante getters/setters), se utilizaron constructores sobrecargados con `this(...)` para centralizar la inicialización y se implementó la sobrecarga de métodos en `actualizarSalario` (porcentaje y monto fijo), respetando una interfaz pública clara. Además, se incorporó un miembro estático (`totalEmpleados`) para llevar el conteo global de instancias y se sobrescribió `toString()` para imprimir el estado de cada empleado de manera legible desde el main, cumpliendo con la consigna solicitada.

Durante la verificación, se instancian objetos con ambos constructores, se ajustan salarios a través de las dos variantes de actualización y se empleó `System.out.println(emp)` para demostrar el funcionamiento de `toString()`.

Anexo

Link repositorio https://github.com/MaquiMarinoni/TUPaD_P2-C22025.git