	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 7

No de Práctica(s): Proyecto 2

Integrante(s): 322044339

322094152

322114461

425093384

322150984

No. de brigada: 6

Semestre: 2026-1

Fecha de entrega: 29 de octubre de 2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

En el día a día, las empresas necesitan llevar un control claro y confiable sobre lo que pagan a su personal, ya sea por un salario fijo o por horas trabajadas. Cuando este proceso se intenta resolver sin una estructura mínima (por ejemplo, mezclando casos en un mismo bloque, sin separar responsabilidades o sin validar entradas) es fácil terminar con cálculos inconsistentes, datos sobreescritos y resultados poco transparentes para el usuario.

Para afrontar este problema, en este proyecto implementamos un Sistema de Nómina en Java que distingue entre empleados asalariados y empleados por horas a partir de una clase abstracta común. Con ello, empleamos y reforzamos los conceptos del paradigma orientado a objetos, haciendo especial uso de la herencia, polimorfismo y encapsulamiento, de modo que cada tipo de empleado calcule sus ingresos según sus propias reglas, pero conserve una interfaz homogénea.

El objetivo es construir una solución sencilla y modular que nos permita: (1) definir qué es un Empleado a nivel conceptual; (2) especializar su comportamiento en subclases concretas (EmpleadoAsalariado y EmpleadoPorHoras); y (3) probar el polimorfismo desde una aplicación principal para verificar que la misma referencia pueda usar distintas implementaciones sin cambios adicionales en el código cliente.

322044339

2. Marco Teórico

2.1. Clases, objetos y clases abstractas

Una clase modela entidades con atributos y comportamientos, mientras que el objeto es su instancia en memoria. Cuando una clase incluye métodos abstractos (sin implementación) y no puede instanciarse, hablamos de una clase abstracta. Su utilidad es definir un contrato común que otras clases concretas deben completar al especializar su comportamiento [3]. En nuestro caso, Empleado actúa como esa base.

2.2. Herencia y polimorfismo

La herencia permite que una clase derive de otra para reutilizar y extender su comportamiento. El polimorfismo posibilita que un mismo método (por ejemplo, ingresos()) se ejecute de forma diferente según el tipo real del objeto que lo invoca, incluso cuando lo referimos a través del tipo de la clase base [4]. Este principio es el que hace que MainApp pueda tratar de manera uniforme a empleados asalariados y por horas.

2.3. Encapsulamiento

El encapsulamiento protege los datos internos de una clase controlando el acceso mediante métodos públicos (getters/setters) y manteniendo los atributos como privados. Esto ayuda a validar coherentemente los estados, a mantener invarianzas y a prevenir modificaciones indebidas [1]. En el proyecto se valida, por ejemplo, que no haya salarios negativos y que las horas estén dentro de un rango aceptable.

2.4. UML: diagramas de clases y de secuencia

UML proporciona representaciones estandarizadas para visualizar diseños. El diagrama de clases sintetiza estructura (atributos, métodos y relaciones), mientras que el diagrama de secuencia muestra la interacción temporal entre objetos para cumplir un escenario [2]. Ambos se incluyen en la Sección 3.5.

322150984

3. Desarrollo

Nuestro proyecto se compone de cuatro clases: Empleado (de tipo abstracta), EmpleadoAsalariado, EmpleadoPorHoras y MainApp. A continuación se describen sus métodos y atributos, asi como propiedades y características tal como fueron implementadas.

3.1. Empleado.java

Empleado es la clase abstracta que concentra los datos generales: nombre, aPaterno y nss. Ofrece constructores, getters/setters y un toString() base. Además, declara el método abstracto ingresos(), que obliga a las subclases a proporcionar su propio cálculo. Esta separación permite hablar de "empleados" de forma genérica, delegando los detalles de pago a cada tipo específico.

3.2. EmpleadoAsalariado.java

EmpleadoAsalariado extiende Empleado y añade el atributo salarioSemanal. Su setter valida que el salario no sea negativo. En ingresos(), devuelve directamente el salario semanal; y en toString() presenta la información completa del empleado con sus ingresos. El diseño es directo porque el pago no depende de horas.

3.3. EmpleadoPorHoras.java

EmpleadoPorHoras extiende Empleado y define salario (por hora) y horas trabajadas. Se validan dos condiciones: (i) salario no negativo; y (ii) rango de horas entre 1 y 180. El cálculo de ingresos() distingue dos tramos: hasta 40 horas se paga tarifa

normal; a partir de la 41, se paga **tiempo extra al doble**. El toString() detalla salario por hora, horas trabajadas e ingresos resultantes.

3.4. MainApp.java

MainApp demuestra el polimorfismo: se declara una referencia Empleado e1 y se le asigna primero un EmpleadoAsalariado y después un EmpleadoPorHoras. En ambos casos, al invocar toString() la ejecución resuelve dinámicamente la versión correspondiente según el tipo real del objeto, lo que confirma el objetivo del diseño.

3.5. Diagramas UML

Diagrama de clases

Figura 1: UML Diagrama de clases

Diagrama de secuencia

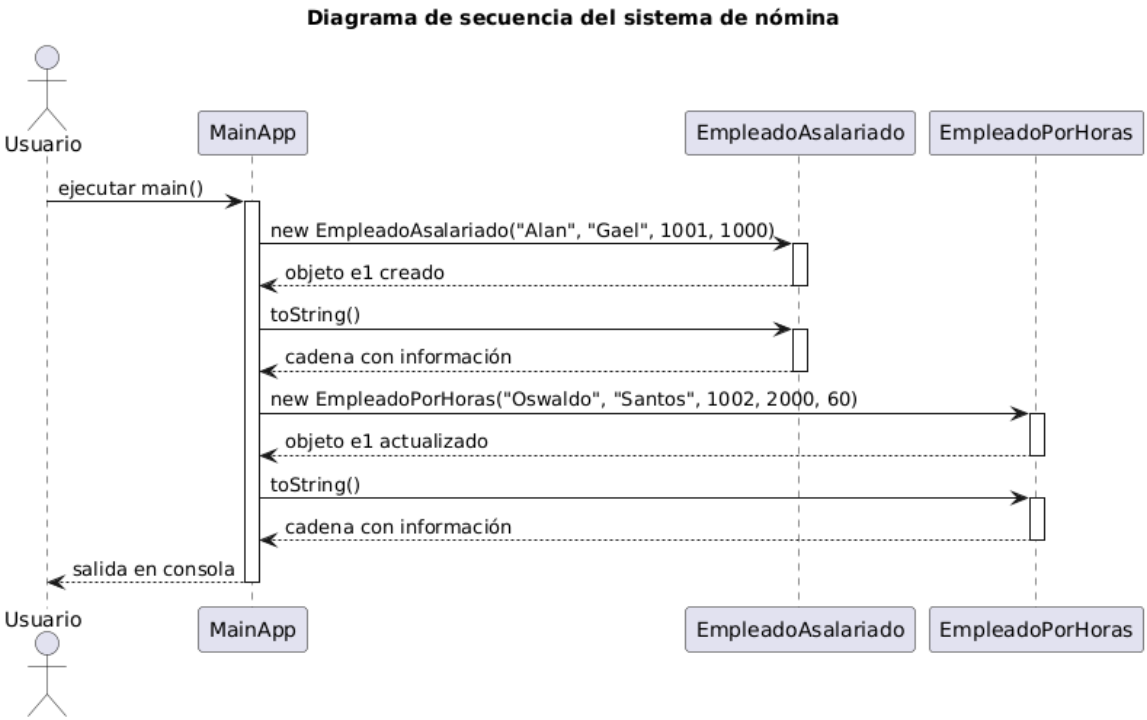

Figura 2: UML Diagrama de secuencia

322094152

4. Resultados

4.1. Empleado asalariado

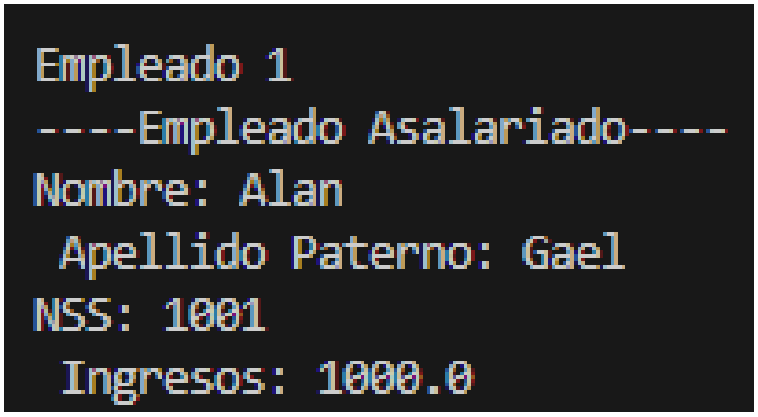

Figura 3: Empleado asalariado

Se creó un empleado con nombre Alan Gael, número 1001 y salario semanal de 1000. Al ejecutar el programa, muestra sus datos y el ingreso calculado.

4.2. Empleado por horas

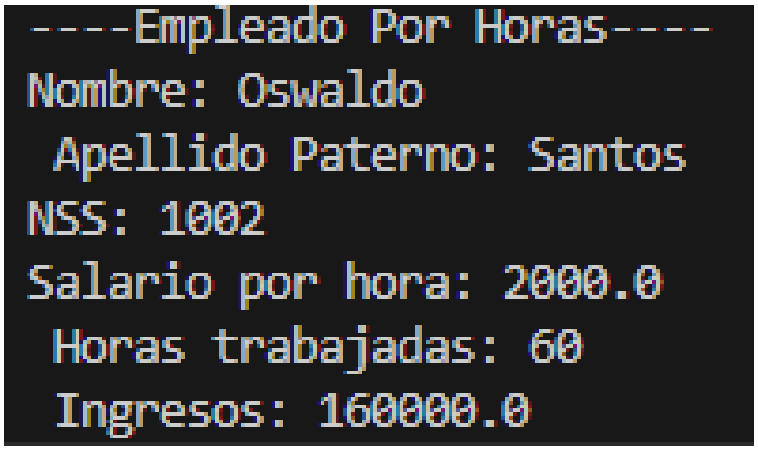

Figura 4: Empleado por horas

Se creó un empleado con nombre Oswaldo Santos, número 1002, salario de 2000 y 60 horas trabajadas. El ingreso total calculado fue de 160000.

322114461

5. Conclusiones

El proyecto consolida el uso de herencia y polimorfismp para resolver un problema cotidiano (la nómina) con una estructura clara: una clase abstracta que define el contrato común y dos subclases que implementan reglas específicas de pago. La validación simple en los setters (no salarios negativos, horas en rango) contribuye al encapsulamiento, permitiendo estados coherentes y mensajes de error oportunos.

En cuanto al desarrollo, separar los casos asalariado y por horas evita mezclar reglas y hace más legible el código. La prueba polimórfica en MainApp demostró que, sin cambiar el programa cliente, la misma referencia puede comportarse según el tipo concreto asociado, que es justo el objetivo práctico de diseñar con clases base y subclases.

En términos generales, los objetivos se cumplieron: se modeló el dominio mínimo necesario, se probó el flujo principal y se documentó con UML (clases y secuencia). Para trabajos futuros, sería natural extender a otros esquemas de pago (por comisión o con bonos) reutilizando la misma arquitectura.

425093384

6. Referencias bibliográficas

Referencias

[1] Picodotdev – Blog Bitix. *Los conceptos de encapsulación, herencia, polimorfismo y composición de la programación orientada a objetos*. Accedido: 29 de octubre de 2025. 2025. URL: https://picodotdev.github.io/blog-bitix/2021/03/los-conceptos-de-encapsulacion-herencia-polimorfismo-y-composicion-de-la-programacion-orientada-a-objetos/.

[2] CEV – Blog. *4 pilares de la Programación Orientada a Objetos*. Accedido: 29 de octubre de 2025. 2025. URL: https://www.cev.com/blog/4-pilares-de-la-programacion-orientada-a-objetos.

[3] Logali Group. *Herencia en la programación orientada a objetos*. Accedido: 29 de octubre de 2025. 2025. URL: https://logaligroup.com/herencia/.

[4] OpenWebinars. *Introducción a POO en Java: Herencia y polimorfismo*. Accedido: 29 de octubre de 2025. 2025. URL: https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/.