



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Ingeniería en computación

Laboratorios de computación salas A y B

Profesor: _____ Ing. Rene Adrian Davila Perez

Asignatura: _____ Programación Orientada a Objetos

Grupo: _____ 1

No. de práctica: _____ 8

Integrante: _____ 424121462

Semestre: _____ 2025-1

Fecha de entrega: _____ 11 de octubre de 2024

Observaciones: _____

CALIFICACIÓN: _____

Índice

1. Introducción (2)
2. Marco Teórico (3)
3. Desarrollo (3-5)
4. Resultados (6-7)
5. Conclusiones (8)
6. Referencias (8)

Introducción:

Basado en los ejercicios realizados en la sesión práctica, se deben completar las siguientes tareas:

- **Ejercicio 1:** Implementación de Algoritmos de Ordenamiento. Escribir un programa en Java que cree una interfaz llamada Ordenamiento. Luego, crear las clases QuickSort y MergeSort que implementen esta interfaz. Cada una de estas clases debe contener el método ordenar, que recibirá un arreglo para ser ordenado usando el algoritmo correspondiente.
- **Ejercicio 2:** Implementación de la Clase Empleado y Clases Derivadas. Crear una clase Empleado con los atributos nombre y rol. Implementar un método calcularSalario que luego será sobrescrito en las clases derivadas. Además, crear las subclases Gerente y Programador, que agregaron sus propios atributos específicos, y sobrescribir el método calcularSalario.

Requisitos Adicionales:

- Los programas deben estar encapsulados correctamente.
- Las clases deben estar organizadas dentro de un paquete.
- Se debe generar un archivo .jar para cada programa.
- La documentación debe generarse utilizando Javadoc.
- El reporte final debe incluir un diagrama de clases que resalte las características orientadas a objetos como la herencia, sobrescritura de métodos, y encapsulamiento.

Este conjunto de ejercicios tiene como objetivo reforzar la comprensión de conceptos fundamentales en programación orientada a objetos, como la herencia y la sobrescritura de métodos, así como la correcta organización del código en paquetes y la creación de documentación profesional.

Se llevará a cabo esta práctica utilizando las mejores herramientas y convenciones posibles, ya que esto no solo mejora la calidad del código, sino que también tiene un impacto en el ámbito laboral. La implementación de comentarios claros y una buena documentación son importantes para facilitar el trabajo en equipo, lo que permite una solución más eficiente de los proyectos.

Marco Teórico:

El polimorfismo permite que un mismo objeto pueda comportarse de diferentes maneras según se requiera o según las necesidades del programador. Permite que las clases se usen de manera intercambiable, lo cual es fundamental para escribir código adaptable a cambios y expansiones.

Cuando se utilizan interfaces, es más fácil agregar nuevas implementaciones sin modificar el código existente. Esto permite mantener el principio de abierto para extensión, cerrado para modificación, un pilar del diseño orientado a objetos.[1] Las interfaces y el polimorfismo permiten trabajar con referencias de clases más generales, lo que fomenta la reutilización de código, adaptándose a las necesidades sin duplicar la lógica de cálculo.

Las interfaces ofrecen una forma de abstracción al definir qué debe hacer una clase, pero no cómo debe hacerlo. Esto facilita la comprensión del sistema, ya que al utilizar una interfaz, los desarrolladores pueden enfocarse en los métodos que una clase necesita cumplir, sin preocuparse por los detalles de implementación.

Utilizar polimorfismo e interfaces fomenta un diseño más claro y estructurado. Los roles y responsabilidades de las clases quedan bien definidos, haciendo que el sistema sea más comprensible para los desarrolladores actuales y futuros.[2]

Desarrollo:

Para la interfaz **Ordenamiento** se tiene:

- Método `ordenar(@arr)`: método sin implementaciones

Para la clase **QuickSort** que implementa **Ordenamiento** se tiene:

- `intercambia(@A, @x, @y)`: método que intercambia elementos de un Array
- `particionar(@A, @p, @r)`: método que particiona un arreglo dado
- `quickSortAplicado(@A, @p, @r)`: método que aplica el algoritmo
- `ordenar(@arr)`: método que aplica el ordenamiento en base al algoritmo

Para la clase **MergeSort** que implementa **Ordenamiento** se tiene:

- `Mezcla(@A, @p, @q, @r)`: método que mezcla subarreglos
- `mergeSortAplicado(@A, @p, @r)`: método que aplica el algoritmo
- `ordenar(@arr)`: método que aplica el ordenamiento en base al algoritmo

El programa ejecuta la ordenación de dos arreglos enteros utilizando los algoritmos de QuickSort y MergeSort. En el método `main`, se definen dos arreglos: {4, 2, 0, 3, 1, 6, 8, 12, 23, 11} y {5, 14, 6, 12, 7, 9, 2, 23, 11, 3}.

Primero, se ordena `arr1` utilizando el algoritmo de **QuickSort** y se imprime el resultado. Luego, se ordena `arr2` con **MergeSort**, también mostrando el arreglo ordenado en pantalla. Para facilitar la visualización de los resultados, se utiliza el método `imprime`, que recorre y muestra cada elemento de los arreglos en la consola.

Para la clase (padre) **Empleado** se tiene:

- nombre: String
- direccion: String
- nombreDeTrabajo: String
- salario: double
- rol: String
- Métodos set y get de cada una de las variables
- Método constructor
- `getBono()`: retorna un double
- `reporteDesempeño(@reporte)`: retorna una cadena vacía
- `manejoDeProyectos(@proyectos)`: retorna una cadena vacía
- `calcularSalario()`: retorna un double 0.0

Para la clase (hija de **Empleado**) **Manager** se tiene:

- Parámetros iguales que los de la clase padre **Empleado**
- Métodos set y get de cada una de las variables
- Método constructor
- `getBono()`: retorna el salario * 0.25. Método sobrescrito
- `reporteDesempeño(@reporte)`: retorna una cadena respecto al desempeño realizado dado la cadena reporte. Método sobrescrito
- `manejoDeProyectos(@proyectos)`: retorna una cadena que muestra proyectos en los que trabaja el manager dada la cadena proyectos. Método sobrescrito
- `calcularSalario()`: retorna la suma del salario + bono. Método sobrescrito

Para la clase (hija de **Empleado**) **Desarrollador** se tiene:

- Parámetros iguales que los de la clase padre **Empleado**
- Métodos set y get de cada una de las variables

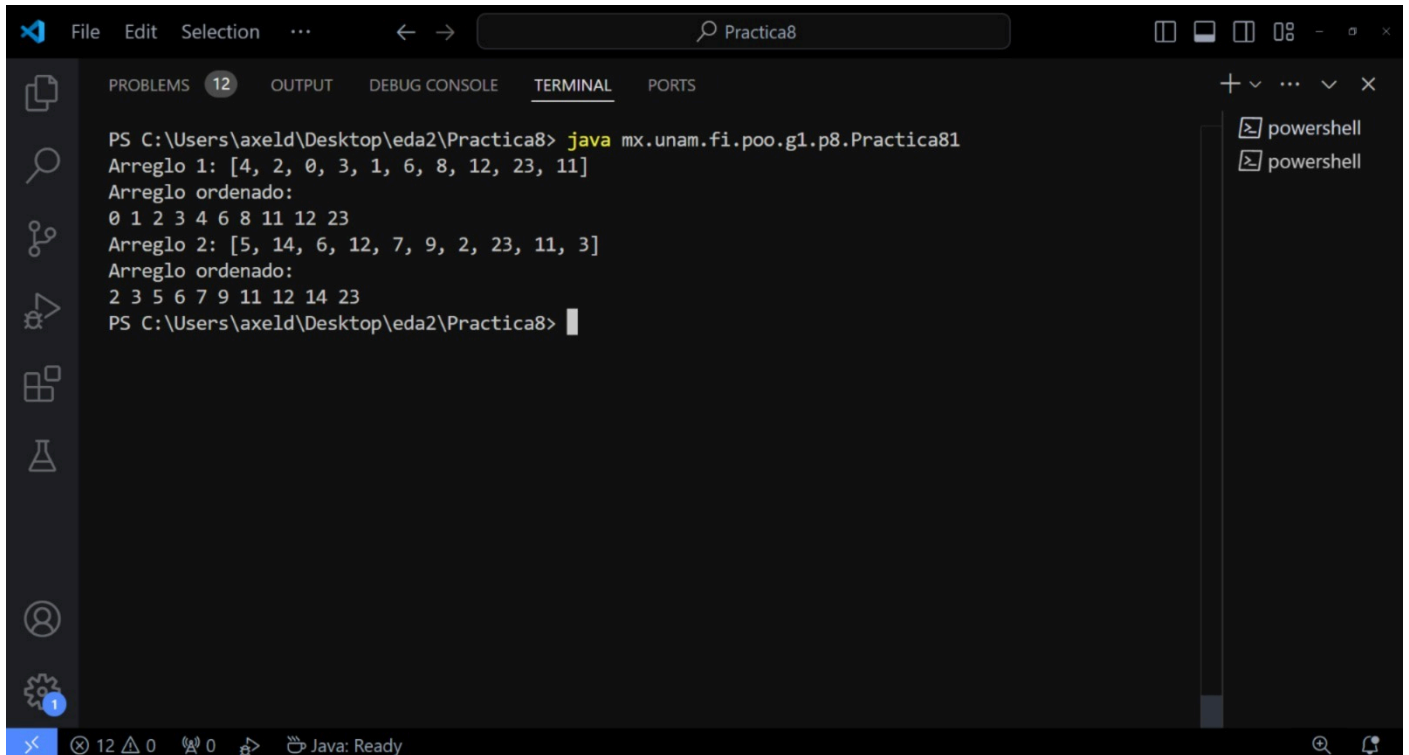
- Método constructor
- getBono(): retorna el salario * 0.15. Método sobrescrito
- reporteDesempeño(@reporte): retorna una cadena respecto al desempeño realizado dado la cadena reporte. Método sobrescrito
- manejoDeProyectos(@proyectos): retorna una cadena que muestra proyectos en los que trabaja el manager dada la cadena proyectos. Método sobrescrito
- calcularSalario(): retorna la suma del salario + bono. Método sobrescrito

Para la clase (hija de **Empleado**) **Programador** se tiene:

- Parámetros iguales que los de la clase padre **Empleado**
- Métodos set y get de cada una de las variables
- Método constructor
- getBono(): retorna el salario * 0.05. Método sobrescrito
- reporteDesempeño(@reporte): retorna una cadena respecto al desempeño realizado dado la cadena reporte. Método sobrescrito
- manejoDeProyectos(@proyectos): retorna una cadena que muestra proyectos en los que trabaja el manager dada la cadena proyectos. Método sobrescrito
- calcularSalario(): retorna la suma del salario + bono. Método sobrescrito

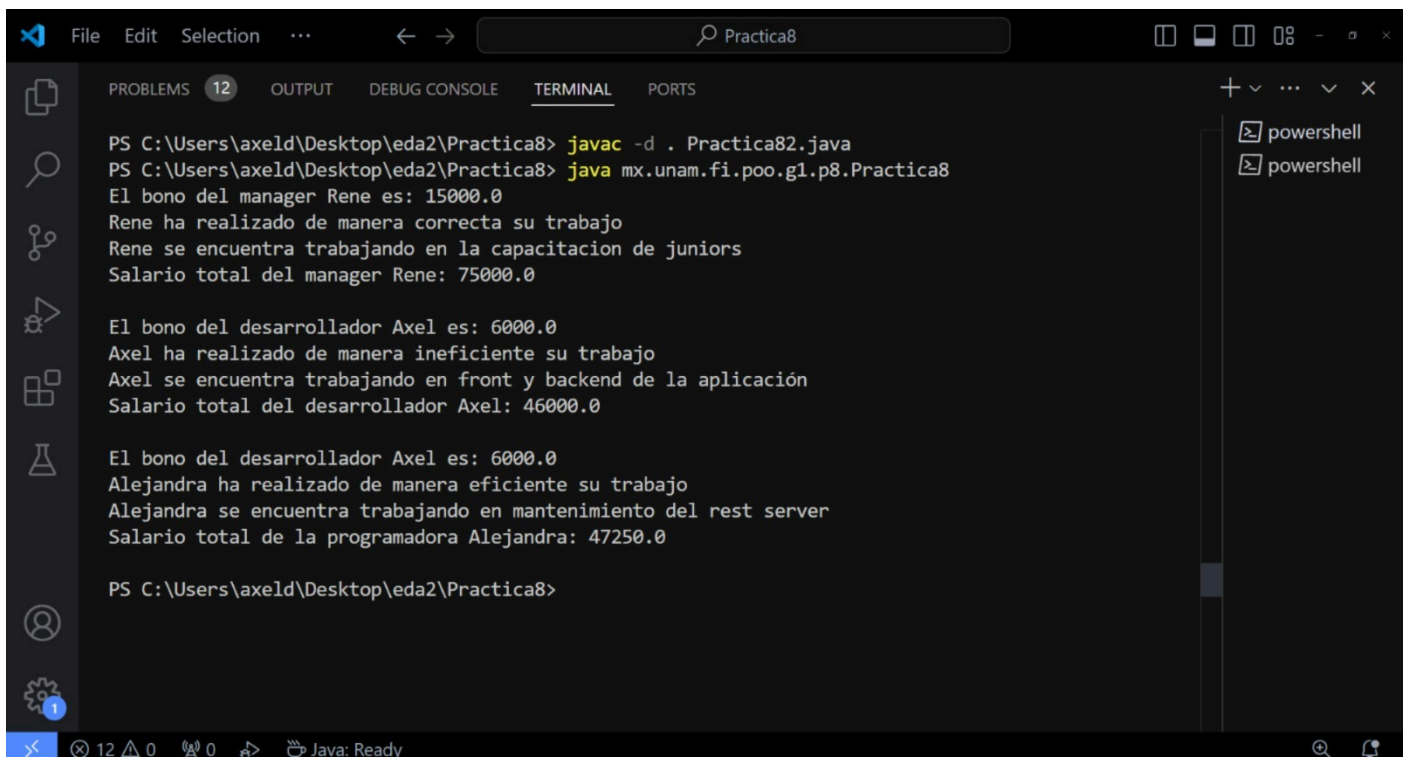
El programa solicita al usuario que ingrese una evaluación de desempeño y un proyecto para los roles de `Manager`, `Desarrollador` y `Programador`. Para el Manager se ingresa "correcta" como evaluación y "capacitación de juniors" como proyecto. Para el Desarrollador se ingresa "ineficiente" y "front y backend de la aplicación". Para la Programadora se ingresa "eficiente" y "mantenimiento del rest server". El programa muestra el bono, la evaluación del desempeño, el proyecto asignado y el salario total para cada rol. Si bien el sistema funciona correctamente, se podrían mejorar las entradas permitiendo opciones predefinidas para las evaluaciones para obtener datos más consistentes y fáciles de procesar.

Resultados:



```
PS C:\Users\axeld\Desktop\eda2\Practica8> java mx.unam.fi.poo.g1.p8.Practica81
Arreglo 1: [4, 2, 0, 3, 1, 6, 8, 12, 23, 11]
Arreglo ordenado:
0 1 2 3 4 6 8 11 12 23
Arreglo 2: [5, 14, 6, 12, 7, 9, 2, 23, 11, 3]
Arreglo ordenado:
2 3 5 6 7 9 11 12 14 23
PS C:\Users\axeld\Desktop\eda2\Practica8>
```

En la imagen se puede ver cómo están los arreglos y su ordenamiento de manera ascendente de acuerdo a los resultados esperados.



```
PS C:\Users\axeld\Desktop\eda2\Practica8> javac -d . Practica82.java
PS C:\Users\axeld\Desktop\eda2\Practica8> java mx.unam.fi.poo.g1.p8.Practica8
El bono del manager Rene es: 15000.0
Rene ha realizado de manera correcta su trabajo
Rene se encuentra trabajando en la capacitacion de juniors
Salario total del manager Rene: 75000.0

El bono del desarrollador Axel es: 6000.0
Axel ha realizado de manera ineficiente su trabajo
Axel se encuentra trabajando en front y backend de la aplicación
Salario total del desarrollador Axel: 46000.0

El bono del desarrollador Axel es: 6000.0
Alejandra ha realizado de manera eficiente su trabajo
Alejandra se encuentra trabajando en mantenimiento del rest server
Salario total de la programadora Alejandra: 47250.0

PS C:\Users\axeld\Desktop\eda2\Practica8>
```

En la imagen se muestra la salida en pantalla de lo ingresado, a su vez también se muestran algunos de los valores definidos en el main para cada una de las clases.

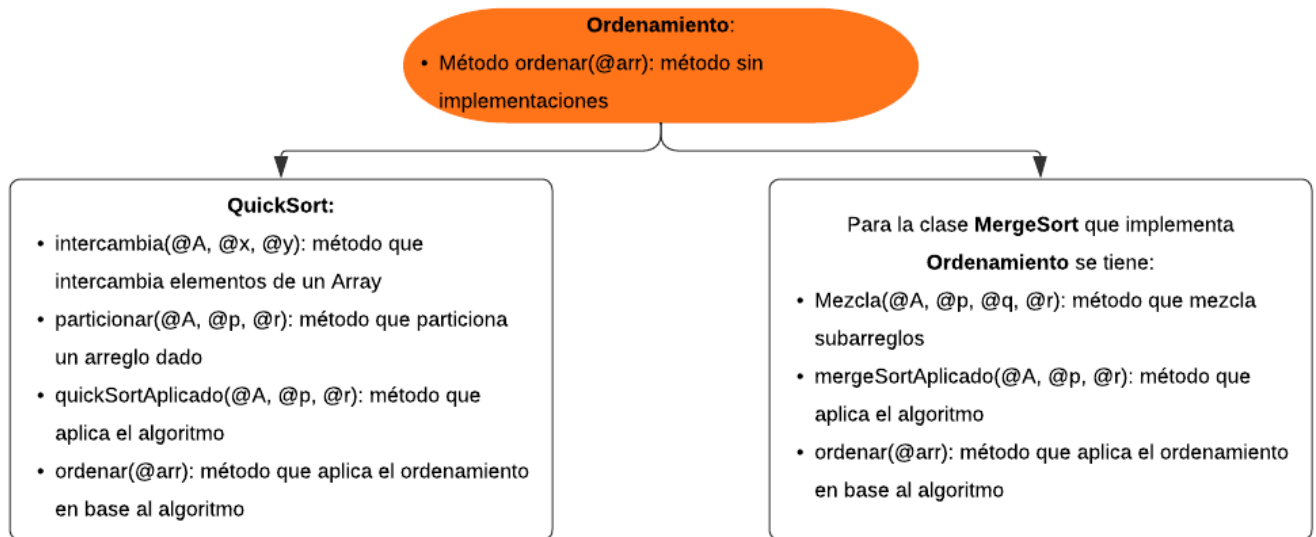


Diagrama de flujo Practica81

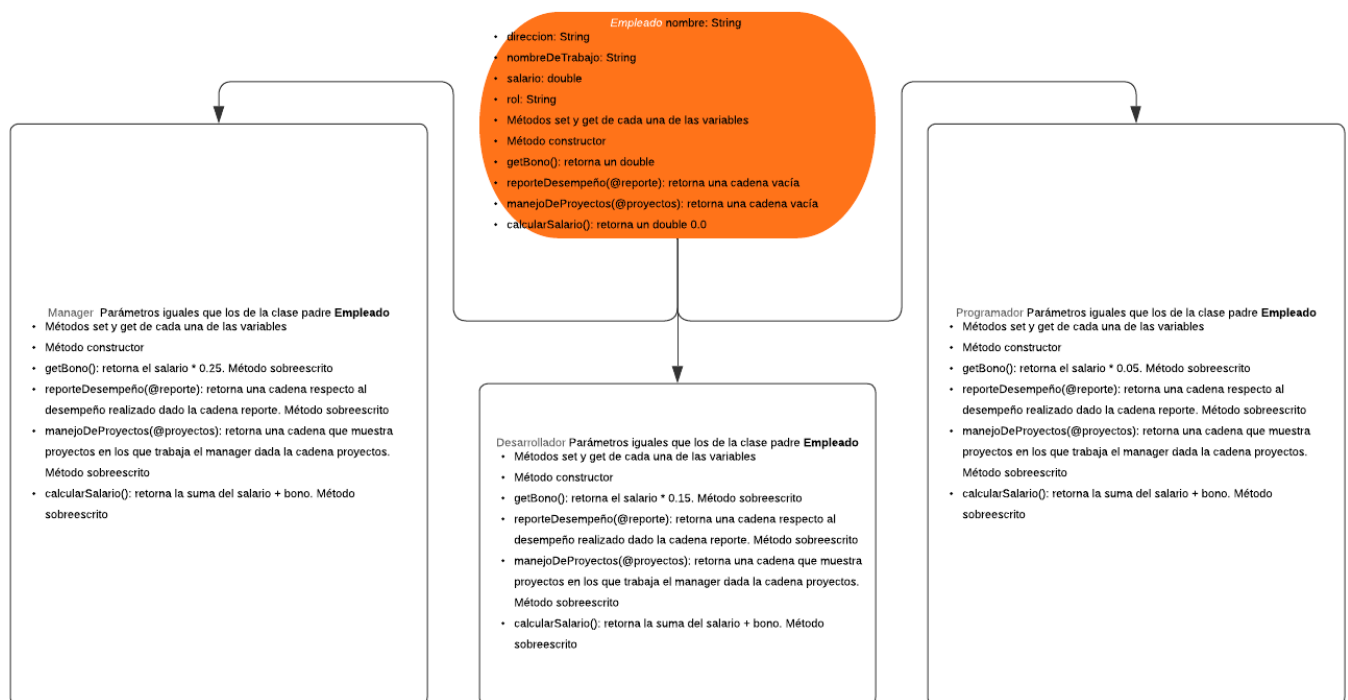


Diagrama de flujo de Practica82

Conclusiones:

La comprensión de los algoritmos de ordenación y el uso de interfaces en Java es fundamental para desarrollar aplicaciones eficientes y bien estructuradas. Estos conceptos permiten implementar diferentes estrategias de ordenamiento, como QuickSort y MergeSort, promoviendo la reutilización del código. Además, la aplicación de principios como la evaluación del desempeño y el manejo de proyectos a través de clases específicas, como Manager, Desarrollador y Programador, ayuda a organizar el código de manera más clara. Sin una base teórica sólida en estos aspectos, la implementación de funcionalidades se vuelve más complicada, lo que podría resultar en un software menos mantenible y difícil de escalar. Por lo tanto, dominar tanto la teoría como la práctica de estos conceptos no solo optimiza el proceso de desarrollo, sino que también eleva la calidad y la legibilidad del software.

Referencias:

- [1] V. Aguilera Perez, "SOLID: Los cinco principios de Diseño Orientado a Objetos (Parte 1)," Medium, 7 de jun., 2024. [En línea]. Disponible: <https://vicente-aguilera-perez.medium.com/solid-los-cinco-principios-de-dise%C3%B1o-orientado-a-objetos-parte-1-2224a84134b1>. [Accedido: Oct. 7, 2024].
- [2] ChatGPT. [En línea]. Disponible: <https://chatgpt.com>. [Accedido: Oct. 7, 2024]. Pregunta: "Platícame más acerca del polimorfismo y las interfaces en java"