	Carátula para entrega de prácticas
Facultad de Ingeniería	**Laboratorio de Docencia**

Laboratorio de Computación Salas A y B

Profesor: René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 1

Número de Práctica: 8

Integrante(s): 321573670

No. de Lista: N/A

Semestre: 2025-1

Fecha de Entrega: 12 / Octubre / 2024

Observaciones:

Calificación:

Índice

1. Introducción **2**

2. Marco Teórico **3**

3. Desarrollo **3**

4. Resultados **7**

5. Conclusiones **23**

Referencias **24**

1. Introducción

- **Planteamiento del problema:**
  Las clases abstractas e interfaces son parte importante de la implementación de polimorfismo en Java, pues nos permiten utilizar la herencia entre clases y así sobrescribir métodos para llevar a cabo dicho concepto, es por ello que, buscando la cimentación del tema, se propuso el desarrollo los siguientes programas:

  - Un programa en el cual se defina la interfaz "Ordenamiento" con el método `ordenar()` el cual será sobrescrito en las clases "BubbleSort" y "SelectionSort" con el fin de ordenar un arreglo.
  - Un programa para el cual se defina la clase abstracta "Figura" con el método `calcularArea()`. Posteriormente definir las clases "Circulo" y "Cilindro" en las cuales se sobrescriba el método definido en "Figura".
  - Un programa en el que se cree la interfaz llamada "Ordenamiento" con la clase `ordenar()`, misma que sea implementada por una clase "QuickSort" y otra clase "MergeSort" con el fin de ordenar dos arreglos.
  - Un programa para la clase abstracta "Empleado" con atributos nombre y rol, además de un método para calcular el salario. De dicha clase deben derivar las clases "Gerente" y "Programador" con atributos propios y sobrescritura del método `calcularSalario()`

- **Motivación:**
  La resolución de los problemas propuestos necesita de la aplicación correcta de los conceptos adquiridos por el alumnado en las clases previas, además que completar esto permitirá cimentar las habilidades y conocimientos del alumnado al respecto.

- **Objetivos:**
  Se espera que el alumnado sea capaz de idear soluciones a los problemas propuestos, haciendo uso de las clases abstractas e interfaces en Java, de este modo profundizando su entendimiento sobre dichos conceptos.

2. Marco Teórico

En Java se puede definir un tipo de clase denominada abstracta, esta es aquella que no puede ser instanciada directamente, pero puede contener tanto métodos abstractos (sin implementación) como métodos con implementación. La finalidad de las clases abstractas es proveer una estructura base para la definición de otras clases haciendo uso de la herencia.[1].

Por otro lado, también podemos definir una interfaz la cual a diferencia de las clases abstractas, las interfaces no pueden contener implementaciones y solo declaran comportamientos, pero las implementaciones de cada uno de los métodos debe de existir en la clase que implemente dicha interfaz. Las interfaces son fundamentales para lograr el polimorfismo y permiten que una clase implemente múltiples comportamientos, rompiendo la restricción de herencia simple [2].

Ambos mecanismos son esenciales para relizar programas basados en POO, proporcionando una base para aplicar el polimorfismo y facilitar la extensión y reutilización de código[2].

3. Desarrollo

- Ejercicio 0:
  Para la resolución del ejercicio propuesto se definió una interfaz llamada "Ordenamiento" con un método denomindado `ordenar()`, misma que fue implementada en las clases "BubbleSort" y "QuickSort" para las cuales se definió una versión del método específica para cada algoritmo de ordenamiento.

  En el método principal, descrito en la clase "Ejercicio0", se definieron dos arreglos y un objeto de cada una de las clases, posteriormente se mandó a llamar el método `ordenar()` para cada algoritmo. Finalmente se definió un método estático para realizar la imperesión del arreglo.

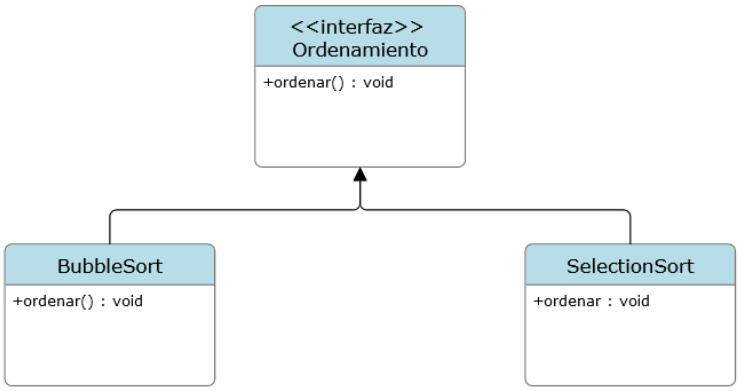


Figura 1: Diagrama UML con interfaz "Ordenamiento" y clases "BubbleSort" y "SelectionSort".

- Ejercicio 1
  En la resolución de este programa se definió una clase abstracta "Figura" con los métodos abstractos: `dibuja()` y `calcularArea()`. Dicha clase fue implementada por la clase "Circulo", para la cual se definió un atributo encapsulado `radio`, un constructor propio y métodos para acceder al atributo mencionado, además se sobrescribieron los métodos definidos en "Figura" para acomodarse a las necesidades de la clase.

  Posteriormente se definió una clase "Cilindro" hija de la clase "Circulo", para dicha clase se definió un atributo encapsulado adicional llamado `altura`, para su constructor se hizo uso del definido en su clase padre pero con la adición del método setter para `altura`. Similarmente a "Circulo" se sobrescribieron los métodos de la clase abstacta.

  Finalmente se definió el método principal en el cual se definieron instancias de ambas clases y, con la ayuda de un método estático definido en la clase prinicpal, se mandó a llamar todas las funcionalidades de cada clase.

- Practica 8.1:
  Para la compleción del programa a desarrollar se comenzó por definir una interfaz "Ordenamiento" con un método `ordenar()` de manera similar a como se hizo para el ejercicio 0.

  Después se definió una clase "QuickSort" que implementa la interfaz ya

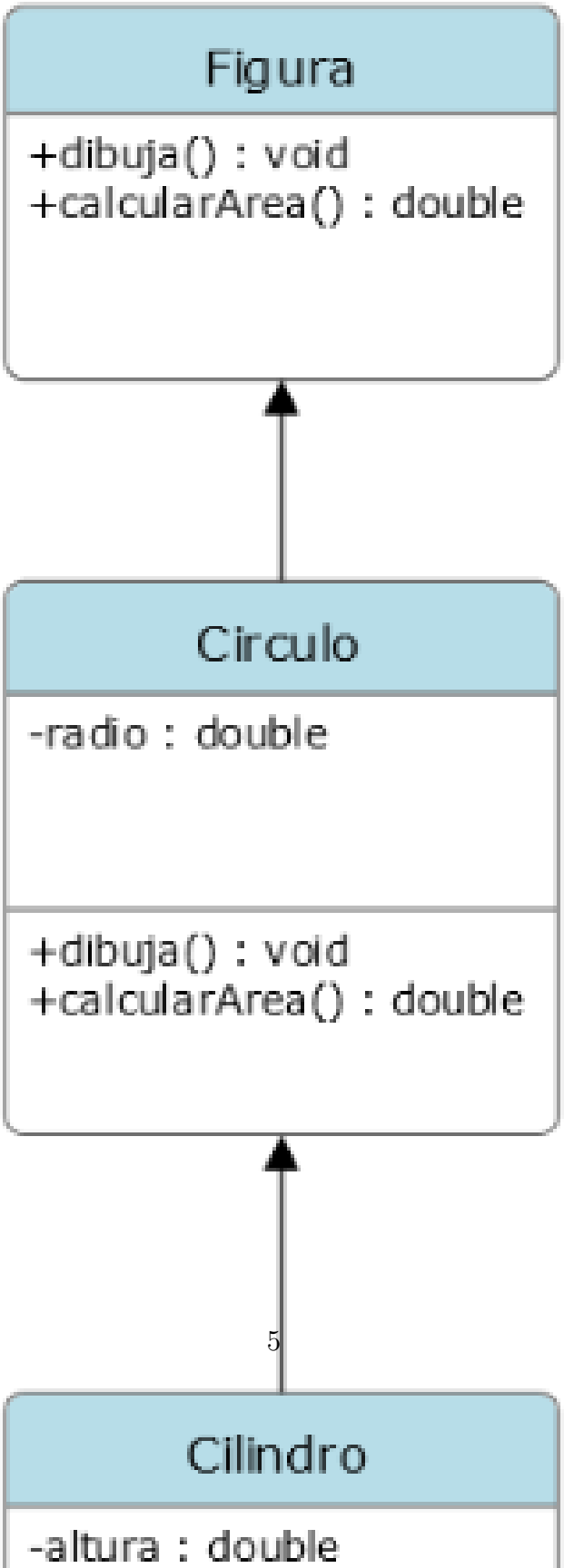


Figura
+dibuja() : void
+calcularArea() : double
Circulo
-radio : double
+dibuja() : void
+calcularArea() : double
Cilindro
-altura : double

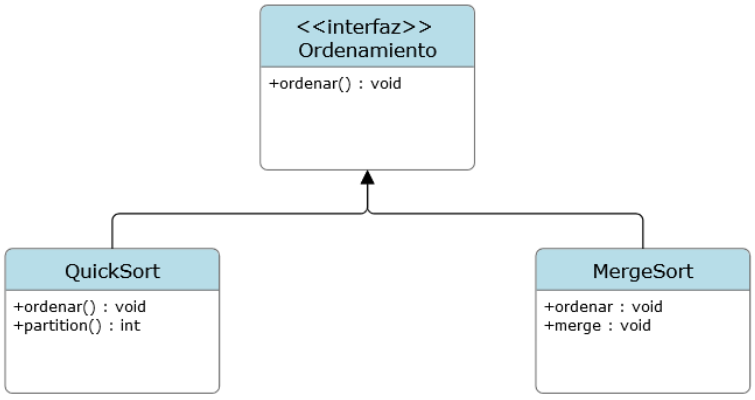


Figura 3: Diagrama UML con interfaz "Ordenamiento" y clases "QuickSort" y "MergeSort".

definida, dentro de dicha clase se sobrescribió el método `ordenar()` con la finalidad de adecuarlo a la lógica del alrgoritmo, el resto de la cual se definió en el método `partition()` de la clase. De manera similar la clase "MergeSort" sobrescribió el método de la interfaz, en su caso también se definió un método `merge()` para completar la lógica.

Finalmente, en la clase principal, se definió el método principal para realizar el ordenamiento en dos arreglos distintos, mismos que se llenan haciendo uso de la función estática definida dentro de la misma clase llamada `llenarArreglo()`, similarmente se definió el método `imprimirArreglo()` para mostrar los resultados obtenidos.

- Practica 8.2:

  En la solución para el último problema propuesto, se definió la clase "Empleado" con atributos encapsulados `nombre` y `rol`, además se definieron los métodos abstractos `calcularSalario()` e `imprimirInformacion()`.

  Posteriormente se definió la clase "Gerente" que extiende a "Empleado", se agregó el atributo encapsulado `numProyectos`, además se sobrescribieron los métodos `calcularSalario()` e `imprimirInformacion()` para los distintos casos de "Gerente".

  De manera similar se definió la clase "Programador", solo que en lugar de un atributo `numProyectos`, se definió el atributo encapsulado `titulo`, igualmente se sobrescribieron los métodos abstractos para

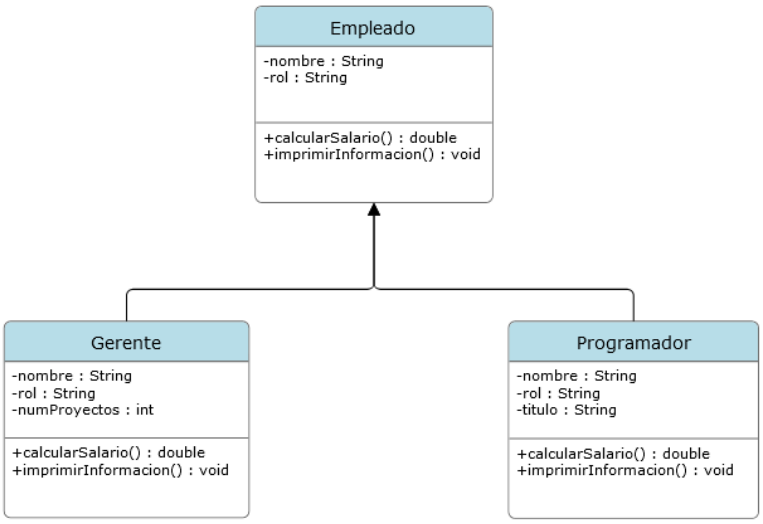


Figura 4: Diagrama UML con clase “Empleado”, “Gerente” y “Programador”.

cumplir con las carácteristicas de la clase.

Finalmente se definió el método principal en el cual se imprime la información de las instancias creadas de cada una de las clases.

4. Resultados

- Ejercicio 0:

  Código:

```
package mx.unam.fi.poo.g1.p8;

interface Ordenamiento{
    void ordenar(int[] arr);
}
```

Figura 5: Definición de la interfaz “Ordenamiento”.

```java
package mx.unam.fi.poo.g1.p8;

public class BubbleSort implements Ordenamiento{
    public void ordenar(int[] arr){
        int n = arr.length;

        for(int i = 0; i < n - 1; i ++){
            for(int j=0; j<n - i - 1; j++){
                if (arr[j]>arr[j+1]){
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```

Figura 6: Lógica para la clase “Bubble Sort” que implementa a la interfaz “Ordenamiento”.

```java
package mx.unam.fi.poo.g1.p8;

public class SelectionSort implements Ordenamiento{
    public void ordenar(int[] arr){
        int n = arr.length;

        for(int i = 0; i < n - 1; i++){
            int minIndex = i;
            for(int j = i + 1; j<n; j++){
                if(arr[j] < arr[minIndex]){
                    minIndex = j;
                }
            }
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

Figura 7: Lógica para la clase “Selection Sort” que implementa a la interfaz “Ordnemiento”.

```java
package mx.unam.fi.poo.g1.p8;

public class Ejercicio0{
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int[] arr = {
            4, 2, 0, 3, 1, 6, 8
        };

        int[] arr1 = {
            6, 7, 3, 8, 4, 1, 5
        };

        System.out.println(x:"Arreglo original:");
        imprime(arr);
        Ordenamiento bubble = new BubbleSort();
        bubble.ordenar(arr);
        System.out.println(x:"Arreglo ordenado con Bubble Sort.");
        imprime(arr);

        System.out.println(x:"Arreglo original:");
        imprime(arr1);
        Ordenamiento selection = new SelectionSort();
        selection.ordenar(arr1);
        System.out.println(x:"Arreglo ordenado con Selecion Sort.");
        imprime(arr1);
    }
```

Figura 8: Método princial en el que se definen los arreglos a ordenar y se crean los objetos para cada clase.

```java
public class Ejercicio0{
    public static void imprime(int[] arr){
        for(int i : arr){
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

Figura 9: Método con el que se imprime el arreglo.

Ejecución:

```
Arreglo original:
4 2 0 3 1 6 8
Arreglo ordenado con Bubble Sort.
0 1 2 3 4 6 8
Arreglo original:
6 7 3 8 4 1 5
Arreglo ordenado con Selecion Sort.
1 3 4 5 6 7 8
```

Figura 10: Ejecución.

- Ejercicio 1:

  Código:

```java
package mx.unam.fi.poo.g1.p8;

abstract class Figura{
    public abstract void dibuja();
    public abstract double calcularArea();
}
```

Figura 11: Definición de la clase abstracta "Figura", en ella se definen dos métodos abstractos `dibuja()` y `calcularArea()`.

```java
package mx.unam.fi.poo.g1.p8;

class Cilindro extends Circulo{
    private double altura;

    public Cilindro(double radio, double altura){
        super(radio);
        setAltura(altura);
    }

    public void setAltura(double altura){
        this.altura = altura;
    }

    public double getAltura(){
        return this.altura;
    }
```

Figura 12: Clase "Cilindro" que extiende a "Circulo", esta tiene sus atributos encapsulados y sus respectivos métodos para acceder y obtener dichos atributos, además se definió un constructor propio.

```java
class Cilindro extends Circulo{

    @Override
    public void dibuja(){
        System.out.println(x:"Un cilindro abstracto");
    }

    @Override
    public double calcularArea(){
        double areaCirculo = super.calcularArea();
        double areaLateral = 2 * Math.PI * getRadio() * getAltura();
        return 2 * areaCirculo * areaLateral;
    }
}
```

Figura 13: Sobrescritura de los métodos `dibuja()` y `calcularArea()` para la clase "Cilindro".

```java
package mx.unam.fi.poo.g1.p8;

class Circulo extends Figura{
    private double radio;

    public Circulo(double radio){
        setRadio(radio);
    }

    public void setRadio(double radio){
        this.radio = radio;
    }

    public double getRadio(){
        return this.radio;
    }
```

Figura 14: Definición de la clase “Circulo” que extiende a “Figura”, tiene definidos los métodos para acceder a su atributo encapsulado.

```java
class Circulo extends Figura{

    @Override
    public void dibuja(){
        System.out.println(x:"Más adelante usará gráficos en Java");
    }

    @Override
    public double calcularArea(){
        return Math.PI * radio * radio;
    }
}
```

Figura 15: Se sobrescriben los métodos `dibuja()` y `calcularArea()` para la clase “Circulo”.

```java
package mx.unam.fi.poo.g1.p8;

public class Ejercicio1{
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Figura circulo = new Circulo(radio:7.0);
        Figura cilindro = new Cilindro(radio:4.0,altura:9.0);

        dibujaCalcula(circulo);
        dibujaCalcula(cilindro);
    }

    public static void dibujaCalcula(Figura figura){
        figura.dibuja();
        double area = figura.calcularArea();
        System.out.println("Area: " + area);
    }
}
```

Figura 16: Método main definido en la clase principal en la cual se crean las instancias de ambas figuras y se ejecutan sus métodos. Además en dicha clase se define un método llamado `dibujaCalcula()`.

Ejecución:

```
Más adelante usará gráficos en Java
Area: 153.93804002589985
Un cilindro abstracto
Area: 22739.56854010988
```

Figura 17: Ejecución del programa.

- Practica 8.1:

  Código:

```java
package mx.unam.fi.poo.g1.p8;


/**
 * Interfaz Ordenamiento
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public interface Ordenamiento{
    /**
     * Método sin definición para implementar el ordenamiento.
     * @param arr -> Arreglo a ordenar.
     * @param izq -> Índice más a la izquiera del arreglo.
     * @param der -> Índice más a la derecha del arreglo.
     */
    void ordenar(int[] arr, int izq, int der);
}
```

Figura 18: Declaración de la interfaz "Ordenamiento", se define el método `ordenar()`.

```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase QuickSort
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class QuickSort implements Ordenamiento {

    /**
     * Método sobrescrito ordenar con el cual se realizan las llamadas recursivas de Quick Sort.
     * @param arr -> Arreglo o sub-arreglo.
     * @param low -> Índice más a la izquiera del arreglo.
     * @param high -> Índice más a la derecha del arreglo.
     */
    @Override
    public void ordenar(int arr[], int low, int high){
        if (low < high){
            int pi = partition(arr, low, high);
            ordenar(arr, low, pi-1);
            ordenar(arr, pi+1, high);
        }
    }
```

Figura 19: Clase "QuickSort" que implementa "Ordenamiento", en dicha clase se sobrescribe el método `ordenar()` para realizar la lógica del algoritmo.

```java
public class QuickSort implements Ordenamiento {
    /**
     * Método con el cual se realiza la partición del arrego en sub-arreglos y se calcula el pivote.
     * @param arr -> Arreglo o sub-arreglo.
     * @param low -> Índice más a la izquiera del arreglo.
     * @param high -> Índice más a la derecha del arreglo.
     * @return i+1 -> Nuevo pivote.
     */
    public int partition(int arr[], int low, int high){
        int pivot = arr[high];
        int i = (low-1);
        for (int j=low; j<high; j++){
            if (arr[j] <= pivot){
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i+1;
    }

}
```

Figura 20: Resto de la lógica para la realización del algoritmo.

```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase MergeSort
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class MergeSort implements Ordenamiento{

    /**
     * Método sobrescrito ordenar en el que se realiza la recursividad.
     * @param arr
     * @param left
     * @param right
     */
    @Override
    public void ordenar(int arr[], int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            ordenar(arr, left, mid);
            ordenar(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }
```

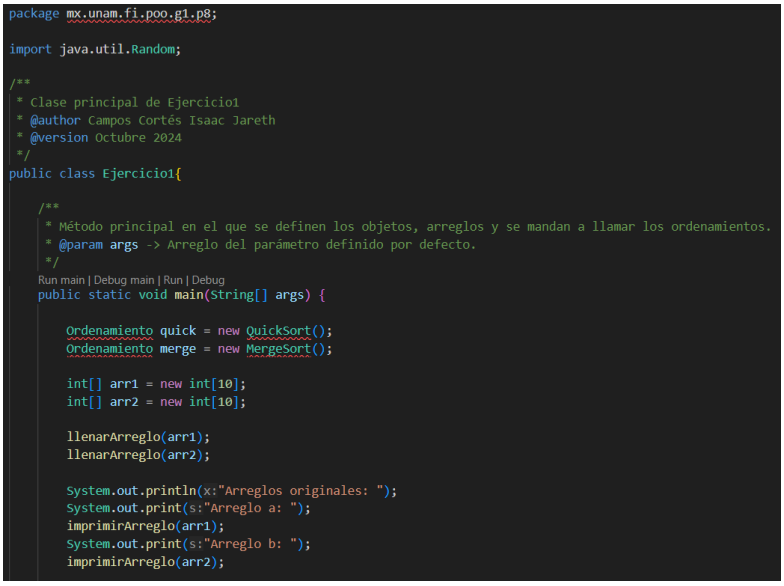
Figura 21: Clase "MergeSort" que implementa a "Ordenamiento", en la misma se sobrescribe el método `ordenar()` para el cual se define parte de la lógica del algoritmo.

```java
public class MergeSort implements Ordenamiento{
    /**
     * Método en el cual se separa el arreglo en sub-arreglos y eventuamente se vuelven a unir.
     * @param arr
     * @param left
     * @param mid
     * @param right
     */
    public void merge(int arr[], int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int L[] = new int[n1];
        int R[] = new int[n2];

        for (int i = 0; i < n1; ++i)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[mid + 1 + j];

        int i = 0, j = 0;

        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
```

Figura 22: Resto del la lógica para el algoritmo merge sort.

```java
public class MergeSort implements Ordenamiento{
    public void merge(int arr[], int left, int mid, int right) {
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }

    }

}
```

Figura 23: Parte final del algoritmo merge sort.



```java
package mx.unam.fi.poo.g1.p8;

import java.util.Random;

/**
 * Clase principal de Ejercicio1
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class Ejercicio1{

    /**
     * Método principal en el que se definen los objetos, arreglos y se mandan a llamar los ordenamientos.
     * @param args -> Arreglo del parámetro definido por defecto.
     */
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {

        Ordenamiento quick = new QuickSort();
        Ordenamiento merge = new MergeSort();

        int[] arr1 = new int[10];
        int[] arr2 = new int[10];

        llenarArreglo(arr1);
        llenarArreglo(arr2);

        System.out.println(x:"Arreglos originales: ");
        System.out.print(s:"Arreglo a: ");
        imprimirArreglo(arr1);
        System.out.print(s:"Arreglo b: ");
        imprimirArreglo(arr2);
```

Figura 24: Clase principal en la que se define el método main, mismo en el que se ejecuta el ordenamiento de dos arreglos llenados aleatoriamente.

```java
public class Ejercicio1{
    public static void main(String[] args) {
        System.out.println(x:"Algoritmo QuickSort en arreglo a: ");
        quick.ordenar(arr1, izq:0, arr1.length-1);
        imprimirArreglo(arr1);

        System.out.println(x:"Algoritmo MergeSort en arreglo b: ");
        merge.ordenar(arr2, izq:0, arr2.length-1);
        imprimirArreglo(arr2);

    }

    /**
     * Método para llenar aleatoriamente el arreglo con números entre 0 y 99
     * @param arr -> Arreglo a llenar.
     */
    public static void llenarArreglo(int[] arr){
        Random random = new Random();
        for (int i = 0; i < 10; i++){
            arr[i] = random.nextInt(bound:99)+1;
        }

    }

    /**
     * Método para realizar la impresión del arreglo.
     * @param arr -> Arreglo a imprimir.
     */
    public static void imprimirArreglo(int[] arr){
        for (int i : arr) {
            System.out.print("["+i+"] ");
        }
        System.out.println(x:"");
    }
}
```

Figura 25: Se definen dos métodos adicionales, uno para llenar el arreglo con valores aleatorios y otro para imprimirlo.

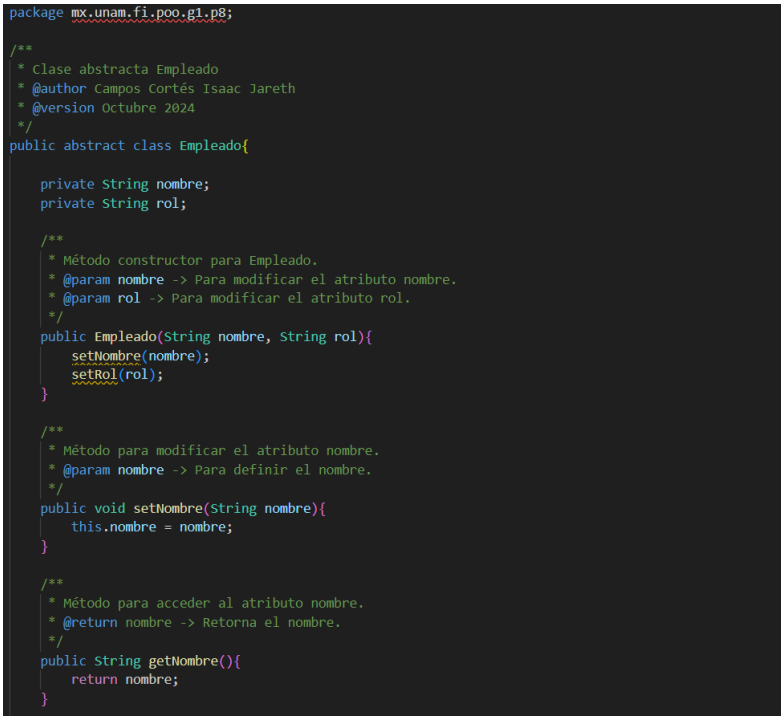
Ejecución:

```
Arreglos originales:
Arreglo a: [62] [25] [33] [28] [58] [80] [67] [32] [13] [44]
Arreglo b: [56] [41] [1] [20] [5] [18] [52] [23] [11] [12]
Algoritmo QuickSort en arreglo a:
[13] [25] [28] [32] [33] [44] [58] [62] [67] [80]
Algoritmo MergeSort en arreglo b:
[1] [5] [11] [12] [18] [20] [23] [41] [52] [56]
```

Figura 26: Ejecución del programa.

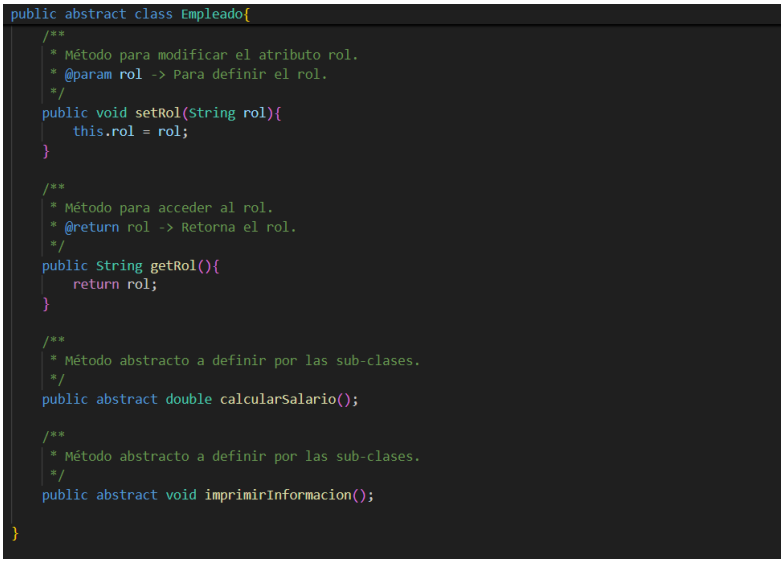
- Practica 8.2:

  Código:



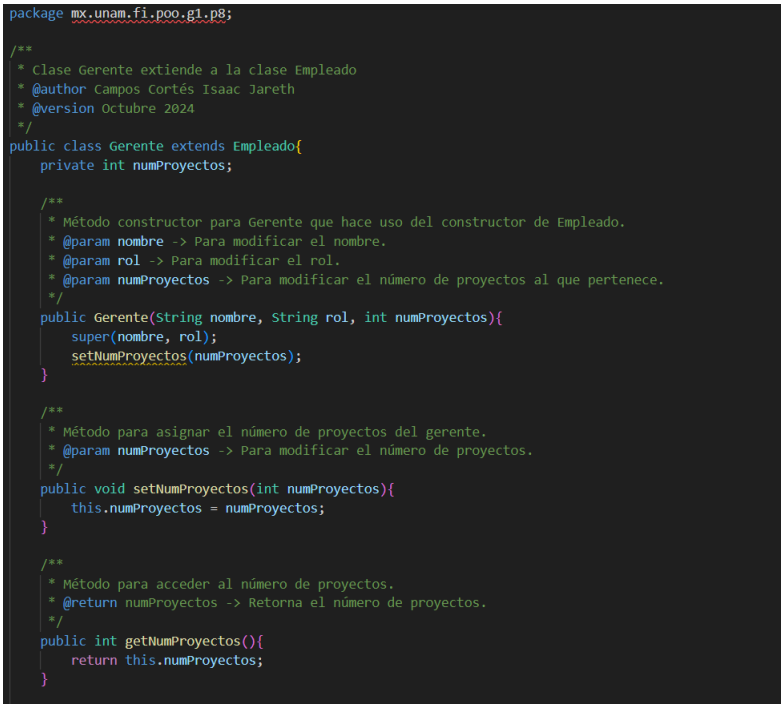
```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase abstracta Empleado
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public abstract class Empleado{

    private String nombre;
    private String rol;

    /**
     * Método constructor para Empleado.
     * @param nombre -> Para modificar el atributo nombre.
     * @param rol -> Para modificar el atributo rol.
     */
    public Empleado(String nombre, String rol){
        setNombre(nombre);
        setRol(rol);
    }

    /**
     * Método para modificar el atributo nombre.
     * @param nombre -> Para definir el nombre.
     */
    public void setNombre(String nombre){
        this.nombre = nombre;
    }

    /**
     * Método para acceder al atributo nombre.
     * @return nombre -> Retorna el nombre.
     */
    public String getNombre(){
        return nombre;
    }
```

Figura 27: Declaración de la clase abstracta “Empleado”, se definen los atributos encapsulados “nombre” y “rol” con sus respectivos setters y getters.



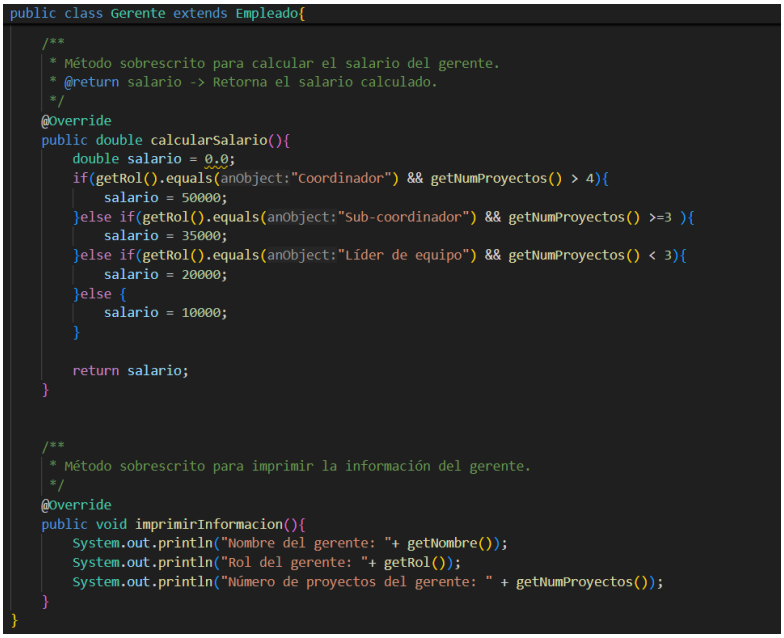
```java
public abstract class Empleado{
    /**
     * Método para modificar el atributo rol.
     * @param rol -> Para definir el rol.
     */
    public void setRol(String rol){
        this.rol = rol;
    }

    /**
     * Método para acceder al rol.
     * @return rol -> Retorna el rol.
     */
    public String getRol(){
        return rol;
    }

    /**
     * Método abstracto a definir por las sub-clases.
     */
    public abstract double calcularSalario();

    /**
     * Método abstracto a definir por las sub-clases.
     */
    public abstract void imprimirInformacion();

}
```

Figura 28: Resto de la clase "Empleado", se definen los métodos abstractos `calcularSalario()` e `imprimirInformacion()`.



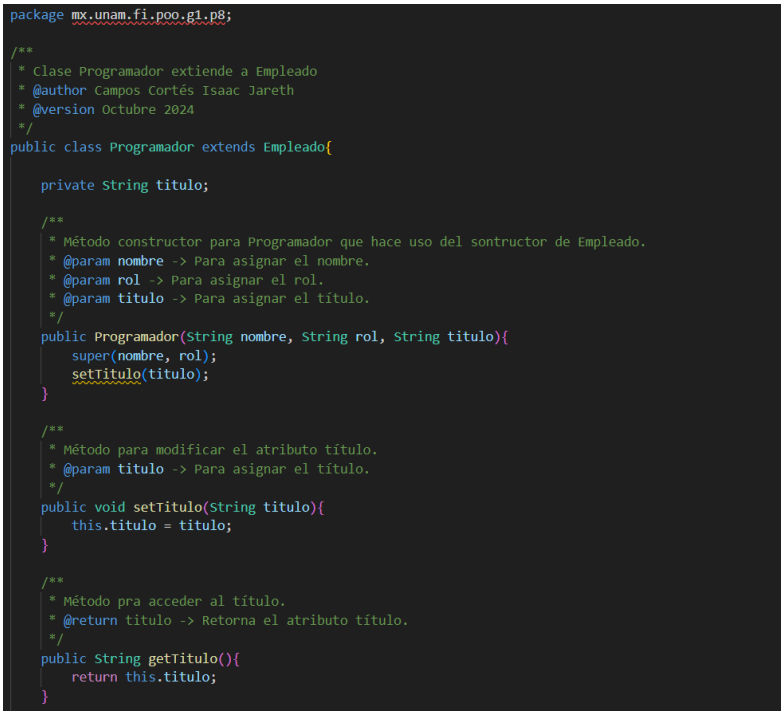
```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase Gerente extiende a la clase Empleado
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class Gerente extends Empleado{
    private int numProyectos;

    /**
     * Método constructor para Gerente que hace uso del constructor de Empleado.
     * @param nombre -> Para modificar el nombre.
     * @param rol -> Para modificar el rol.
     * @param numProyectos -> Para modificar el número de proyectos al que pertenece.
     */
    public Gerente(String nombre, String rol, int numProyectos){
        super(nombre, rol);
        setNumProyectos(numProyectos);
    }

    /**
     * Método para asignar el número de proyectos del gerente.
     * @param numProyectos -> Para modificar el número de proyectos.
     */
    public void setNumProyectos(int numProyectos){
        this.numProyectos = numProyectos;
    }

    /**
     * Método para acceder al número de proyectos.
     * @return numProyectos -> Retorna el número de proyectos.
     */
    public int getNumProyectos(){
        return this.numProyectos;
    }
```

Figura 29: Código de la clase “Gerente” la cual extiende a “Empleado”. Se define un constructor y se agrega un nuevo atributo encapsulado.



```java
public class Gerente extends Empleado{

    /**
     * Método sobrescrito para calcular el salario del gerente.
     * @return salario -> Retorna el salario calculado.
     */
    @Override
    public double calcularSalario(){
        double salario = 0.0;
        if(getRol().equals(anObject:"Coordinador") && getNumProyectos() > 4){
            salario = 50000;
        }else if(getRol().equals(anObject:"Sub-coordinador") && getNumProyectos() >=3 ){
            salario = 35000;
        }else if(getRol().equals(anObject:"Líder de equipo") && getNumProyectos() < 3){
            salario = 20000;
        }else {
            salario = 10000;
        }

        return salario;
    }


    /**
     * Método sobrescrito para imprimir la información del gerente.
     */
    @Override
    public void imprimirInformacion(){
        System.out.println("Nombre del gerente: "+ getNombre());
        System.out.println("Rol del gerente: "+ getRol());
        System.out.println("Número de proyectos del gerente: " + getNumProyectos());
    }
}
```

Figura 30: Se sobrescriben los métodos `calcularSalario()` e `imprimirInformacion()`.



```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase Programador extiende a Empleado
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class Programador extends Empleado{

    private String titulo;

    /**
     * Método constructor para Programador que hace uso del sontructor de Empleado.
     * @param nombre -> Para asignar el nombre.
     * @param rol -> Para asignar el rol.
     * @param titulo -> Para asignar el título.
     */
    public Programador(String nombre, String rol, String titulo){
        super(nombre, rol);
        setTitulo(titulo);
    }

    /**
     * Método para modificar el atributo título.
     * @param titulo -> Para asignar el título.
     */
    public void setTitulo(String titulo){
        this.titulo = titulo;
    }

    /**
     * Método pra acceder al título.
     * @return titulo -> Retorna el atributo título.
     */
    public String getTitulo(){
        return this.titulo;
    }
```

Figura 31: Clase “Programador” que extiende a “Empleado”, se define su contructor, y los setters y getters para su atributo encapsulado adicional.

```java
public class Programador extends Empleado{
    /**
     * Método sobrescrito para calcular el salario del Programador.
     * @return salario -> Retorna el salario calculado.
     */
    @Override
    public double calcularSalario(){
        double salario = 0.0;

        if(getRol().equals(anObject:"Full Stack") && getTitulo().equals(anObject:"Senior")){
            salario = 50000;
        }else if(getRol().equals(anObject:"Backend") && getTitulo().equals(anObject:"Middle")){
            salario = 35000;
        }else if(getRol().equals(anObject:"Frontend") && getTitulo().equals(anObject:"Junior")){
            salario = 20000;
        }else {
            salario = 10000;
        }

        return salario;
    }

     /**
     * Método sobrescrito para imprimir la información del programador.
     */
    @Override
    public void imprimirInformacion(){
        System.out.println("Nombre del programador: "+ getNombre());
        System.out.println("Rol del programador: "+ getRol());
        System.out.println("Título del programador: " + getTitulo());
    }
}
```

Figura 32: Se sobrescriben los métodos `calcularSalario()` e `imprimirInformacion()` para tener funcionalidad específica de la clase.

```java
package mx.unam.fi.poo.g1.p8;

/**
 * Clase principal para Ejercicio2
 * @author Campos Cortés Isaac Jareth
 * @version Octubre 2024
 */
public class Ejercicio2 {
    /**
     * Método principal en el que se ejecuta la funcionalidad del programa.
     * @param args -> Arreglo definido por defecto para main.
     */
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Gerente gerente = new Gerente(nombre:"Rodrigo Pérez", rol:"Coordinador", numProyectos:5);
        Programador programador = new Programador(nombre:"Dolores Salgrado", rol:"Backend", titulo:"Middle");

        System.out.println(x:"A continuación se presenta la información sobre dos empleados:");
        System.out.println(x:"\nInformación sobre el gerente: ");
        gerente.imprimirInformacion();
        System.out.println(x:"\nInformación sobre el programador: ");
        programador.imprimirInformacion();
        System.out.println(x:"");


    }
}
```

Figura 33: Definición del método main del programa.

Ejecución:

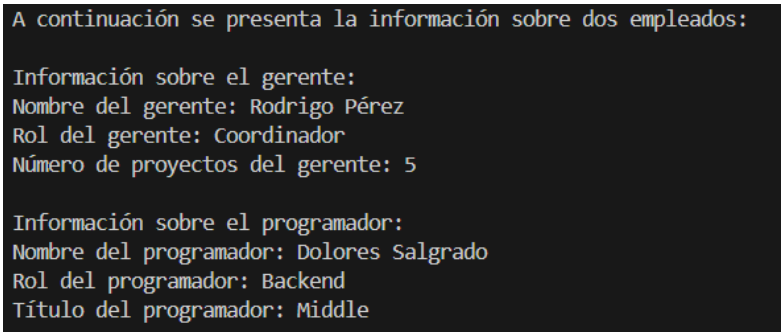


Figura 34: Ejecución del programa.

5. Conclusiones

La práctica realizada nos permitió comprender de manera mucho más tangible y comprensiva el funcionamiento de las clases abstractas y las interfaces al momento de desarrollar un programa en Java y al mismo tiempo idear situaciones en las que su aplicación resulta útil, esto gracias a la correcta compleción de los programas propuestos.

Referencias

[1] P. Deitel y H. Deitel, *Java How to Program*, 11th. Pearson, 2017.

[2] J. Bloch, *Effective Java*, 3rd. Addison-Wesley, 2018.