| | |
|---|---|
| | **Carátula para entrega de prácticas** |
| Facultad de Ingeniería | Laboratorios de docencia |

# Laboratorio de Computación Salas A y B

*Profesor(a):* _____

*Asignatura:* _____

*Grupo:* _____

*No de Práctica(s):* _____

*Integrante(s):* 321172974

_____

_____

_____

*No. de lista o brigada:* _____

*Semestre:* _____

*Fecha de entrega:* _____

*Observaciones:* _____

_____

CALIFICACIÓN: _____

# ÍNDICE

# Introducción

## Planteamiento del problema

Diseñar e implementar clases que resuelvan los siguientes planteamientos:

1. Programa que cree una interfaz Ordenamiento y a partir de ella, 2 clases de ordenamiento: QuickSort y MergeSort que implementen dicha interfaz
2. Programa que contenga la clase base Empleado con atributos nombre y rol, así como con el método calcular salario; derivadas de ella la clase Gerente y Programador con sus propios atributos y que sobreescriban el método calcular salario.

## Motivación

La abstracción y el Polimorfismo son 2 conceptos fundamentales y pilares de la Programación Orientada a Objetos, ponerlos en práctica con clases abstractas y Polimorfismo en objetos nos permitirá apreciar mejor su importancia e impacto en los programas realizados con ellos.

## Objetivos

- En las implementaciones de los programas solicitados, utilizar **interfaces** y **clases abstractas**. Lograr entender, por lo tanto, las situaciones en las que ambos conceptos son útiles
- Se aprecie el **Polimorfismo** en los programas realizados y se logre comprender su importancia
- Aplicar conceptos pasados de la Programación Orientada a Objetos para tener un aplicación completa, esto es: *contar con Encapsulamiento en clases, clases contenidas en paquetes, archivo JAR, documentación javadocs e incluir diagramas de clases (UML).*

# Marco Teórico

## Herencia

Cuando se crea una clase, podemos optar por en vez de declarar miembros completamente nuevos, designar que la nueva clase herede los miembros de una clase existente; en otras palabras, la nueva subclase **extenderá** a la superclase. A este concepto de extender se le conoce como **herencia.**

A la nueva subclase, se le pueden añadir más campos y métodos, de tal forma que dichos objetos tengan dichas capacidades, además de las heredadas.

### `Herencia jerárquica`

En esta herencia, una clase sirve como la **superclase,** siendo la madre de las que la extienden, las que extienden a la superclase se llaman **subclases.**

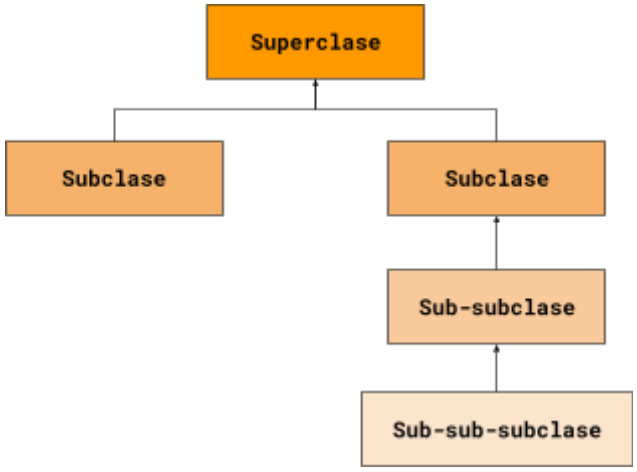



*Figura 1. Jerarquía de herencia o jerarquía de clases*

Esto significa que una clase padre puede tener múltiples subclases que heredan sus características y comportamientos. **[1, p.414]**

### Super

Palabra reservada que hace referencia a la clase base o superclase **[2]**

Se puede usar para **invocar el constructor de la clase base**

```
super(argumentos del constructor de clase base)
```

Para **invocar métodos de la clase base**

```
super.metodoBase()
```

Para **hacer referencia a atributos de la clase base**, útil cuando hay ambigüedad por nombres

```
super.atributo
```

### Herencia "Múltiple" con Interfaces

*"Cuando una clase puede tener más de una superclase y heredar características de todas la clases principales"* **[2]**

Aunque Java no soporta la herencia Múltiple como tal, en el sentido que una clase no puede heredar (extender) más de 1 clase, existe el concepto de interfaz, que permite emular este tipo de herencia.

**Sintaxis de extensión (herencia simple / jerárquica):**

```
... class subclase extends superclase { ... }
```

**Sintaxis de implementación de interfaz (simulación de herencia múltiple):**

```
... class clase implements interfaz { ... }
```

## Polimorfismo

*El polimorfismo nos permite "programar en forma general"* **[1, p.418]**

Considérese una plantilla (clase) de un objeto, por ejemplo, Animal, se sabe que aparte de sus características, los animales tienen acciones (métodos) o comportamientos, dichos comportamientos pueden ser comunes, un animal puede ***moverse*** por ejemplo, pero no todos los animales lo hacen de la misma manera.

*El procedimiento de dicho comportamiento es diferente para **distintas formas** del objeto Animal.*

De esta forma, un programa puede enviar el mismo mensaje, el de moverse, para un conjunto de Animales, pero cada animal dependiendo de la forma que este tenga (la subclase específica) lo hará de cierta forma.

*"El **concepto clave del polimorfismo** es confiar en que cada objeto sepa 'como hacer lo correcto' en respuesta a la llamada del mismo método."* **[1, p.418]**

Un objeto puede ser muy general y tener comportamientos generales, que pueden o no definirse. Al momento de que una subclase la extiende, hereda naturalmente dichos comportamientos, pero los mismos pueden ser **sobrescritos** para que se comporten de la manera en que lo requiera la clase específica donde reside.

### Tipos de Polimorfismo

**[2]**

| En tiempo de Compilación | En tiempo de Ejecución |
|---|---|
| Conocido también como enlace estático, anticipado o sobrecarga | Conocido también como enlace dinámico, tardío o sobreescritura |
| Ejecución rápida | Ejecución lenta |
| Herencia no involucrada | Herencia involucrada |
| **Sobrecarga de Métodos,** cuando más de un método comparte el mismo nombre con diferentes parámetros o formas y diferentes tipos de retorno **[2]** | **Sobreescritura de métodos**, cuando una clase derivada tiene una definición para uno de los métodos miembro de la clase base **[2]** |

## Polimorfismo y herencia en acción

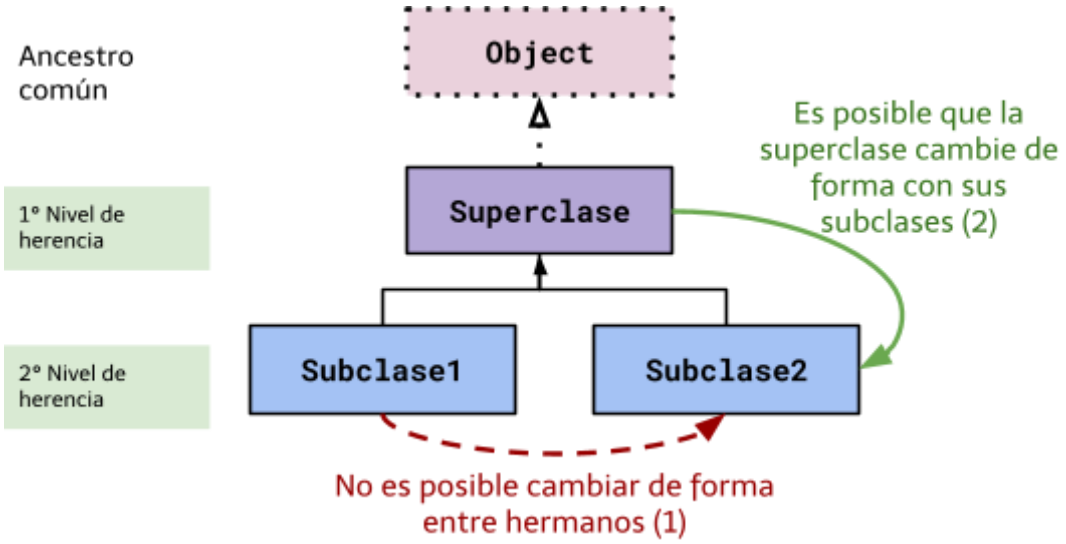



*Figura 2. Ejemplo de Polimorfismo y herencia*

Un **ejemplo general de Polimorfismo** es cuando una **superclase** adquiere las distintas formas de sus **subclases**:

```
Superclase s = new Subclase1();
```

```
s = new Subclase2();
```

*La reasignación es válida dado que s es del tipo superclase, la cuál puede cambiar de formas a las de sus subclases.*[1]

En la *Figura 2* se puede apreciar una jerarquía de clases de manera general: una superclase se extiende hacia 1 o más subclases y dichas subclases pueden extenderse a otras, creando **niveles de herencia**, donde se parte de una superclase madre de todo y se va desglosando o profundizando en la herencia. Se puede apreciar en la figura los 2 niveles de herencia que tiene dicho diagrama. *(La clase ancestro Object no se cuenta en los niveles de herencia)*

En Java, toda clase se extiende de la clase **`Object`**, siendo esta *el ancestro común de todas y la única que no tiene superclase*. [2] Cada clase es implícitamente una subclase de la clase **`Object` [2]**

[1] **Importante enfatizar** que cuando se hace un cambio de forma, los métodos y atributos no heredados de *subclase1* no serán accesibles una vez se haya hecho el cambio al tipo *subclase2*.
[2] Se recuerda que una clase tiene una y solo una superclase directa

También, en (1) una clase dada no puede cambiar a la forma de alguna clase hermana del mismo de herencia, en el ejemplo entonces no se podría:

```
Subclase1 s1 = new Subclase2();
```

Pero, en (2), una clase padre su puede adquirir la forma de alguna de sus clases hijas

```
Superclase ss = new Subclase2();
```

## Clases Abstractas

Al momento de declarar una clase, se puede requerir que algún método no tenga definición en dicho momento, dado que es muy general dicho comportamiento para el nivel de herencia en el que se encuentra, por lo que se puede emplear el tipo de clase abstracta:

```
abstract class
```

que definirá 1 o más métodos abstractos usando la palabra reservada `abstract` , ejemplo:

```
public abstract void area()
```

Estas clases, dado que declaran un método no concreto, **no son instanciables** y **esperan que sean extendidas** y alguna de sus **subclases defina dicho(s) métodos para convertirlos en métodos concretos y puedan ser instanciables**.

De tal forma que las subclases de una superclase abstracta, alguna, debe definir las piezas faltantes, de otra forma habrá error de compilación.

## Desarrollo

### Ejercicios en clase

| `Ejercicio0` | |
|---|---|
| **Planteamiento** | **Solución** |
| Programa que implementa una interfaz Ordenamiento con el método ordenar para las clases BubbleSort y SelectionSort | Se define la interfaz: `Ordenamiento`, misma que cuenta con 1 solo método:<br><br>`ordenar(int[ ] arr);` |

| | |
|---|---|
| | Se definen las clases `BubbleSort` y `SelectionSort` implementando la interfaz antes declarada, ahora implementando aquel método ordenar siguiendo el algoritmo respectivo para cada algoritmo [3]<br><br>En la clase principal estática, se define un arreglo de prueba, se define también mediante Polimorfismo 2 instancias de `BubbleSort` y `SelectionSort`. Note que las interfaces no son instanciables, sin embargo las clases que las implementan si, por ende la línea:<br><br>`Ordenamiento bubble = new BubbleSort();`<br><br>es válida.<br><br>Se define también un método imprime para poder imprimir cualquier arreglo<br><br>Finalmente, se llama al método de ordenamiento para cada objeto y se visualiza que para ambos, el arreglo de prueba, originalmente desordenado, es ordenado por ambos algoritmos implementados en sus respectivas clases. |

| `Ejercicio1` | |
|---|---|
| **Planteamiento** | **Solución** |
| Programa que contenga la clase abstracta Figura con el método calcular área. A partir de ella crear las clases Círculo y Cilindro que definan/sobreescriban el método calcular área | Se define la clase abstracta `Figura`, la cuál define los métodos abstractos dibuja y `calculaArea` que serán posteriormente definidos en las **clases concretas**. |

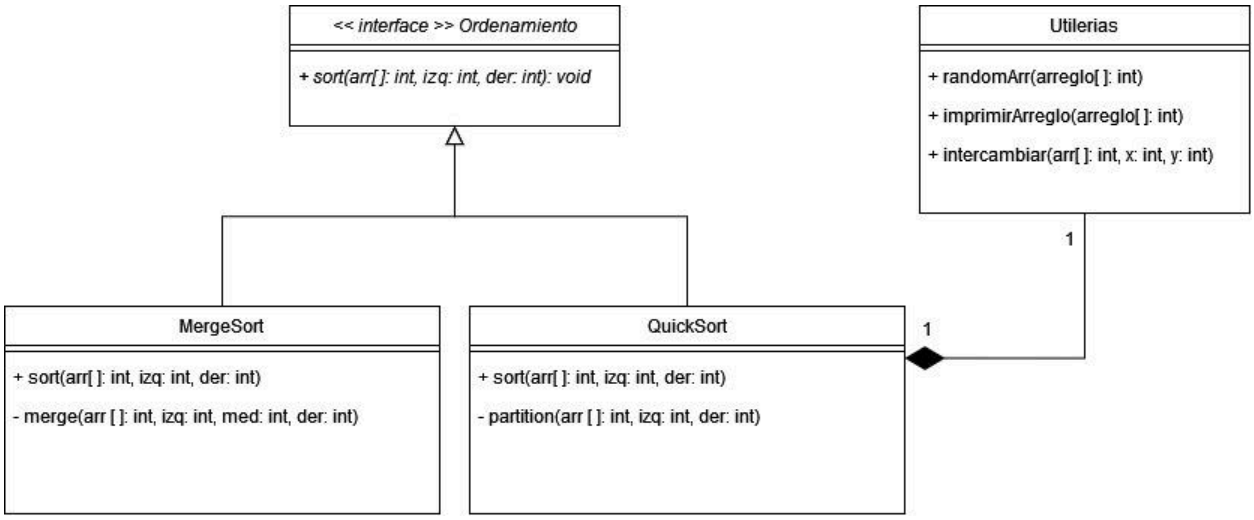
[3] Una descripción general de ambos algoritmos, **BubbleSort** y **SelectionSort**, se encuentra en el **Apéndice A**.

| | |
|---|---|
| | Se define una primera clase, `Circulo`, que hereda de `Figura`, encapsula su atributo radio y sobreescribe los métodos declarados en la superclase: `dibuja()` y `calculaArea()`, dandoles una definición concreta.<br><br>Se define una segunda clase, `Cilindro`, que hereda de `Figura`.<br><br>Se le da implementación a los métodos abstractos de la superclase y encapsula sus atributos privados<br><br>En una clase estática, se definen ejemplos para poder ver el funcionamiento de sus métodos para cada objeto declarado y sus diferentes formas |

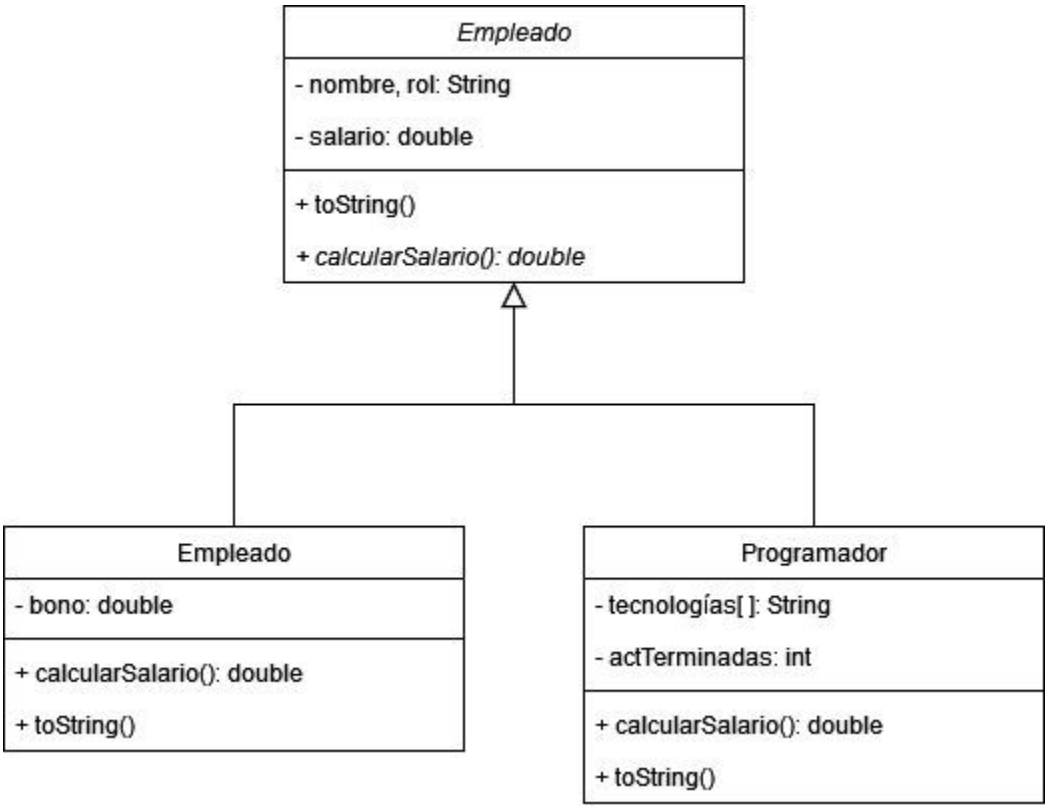
## Ejercicios de Tarea

### Diagramas UML

### Ejercicio 1



## Ejercicio 2




## Explicación de Ejercicios

### Ejercicio1

Se elaboró un archivo adicional:

`Utilerias.java`

Con algunas funciones de uso común para ambos algoritmos de ordenamiento, como lo son:[4]

| | |
|---|---|
| `public void randomArr(int[] arreglo)` | Llena el arreglo indicado con números aleatorios de 1 – 1000 |
| `public void imprimirArreglo(int[]` | Imprime el arreglo indicado |

[4] Note que aunque los métodos son instanciables, se pudo haber implementado una clase estática, dado que los datos no dependen de la instancia de un objeto, sin embargo, dado que se solicita no usar clases estáticas y que en los requerimientos de dicho ejercicio se solicita emplear una interfaz, para mayor simplicidad, se generó finalmente una clase instanciable

| | |
|---|---|
| `arreglo)` | |
| `public void intercambiar(int[] arr, int x, int y)` | Intercambia el valor de 2 variables a y b, siendo ambas parte del arreglo arr |

Cada clase de ordenamiento debe demostrar que ordena, por lo tanto se declaran 2 arreglos distintos (dado que uno será ordenado antes), mismos que para generar sus elementos aleatorios, hará uso de `randomArr();`
Sobre dichos arreglos se hará el ordenamiento respectivo.

**Ejercicio 2**

Para este ejercicio, se implemento la **clase abstracta** Empleado como base, la cuál tiene como método principal abstracto: calcularSalario().
Esta clase se extiende a 2 subclases, ambas en el mismo nivel de herencia y que definen al método de calcularSalario de acuerdo a sus necesidades, para esto se establece la siguiente convención:

| Gerente | Programador |
|---|---|
| CalcularSalario se obtiene como:<br><br>salario (base) + bono<br><br>Donde ambos datos son pasados por el constructor al momento de instanciar un objeto de este tipo.<br><br>*El bono se puede incrementar o decrementar de acuerdo a la necesidad* | CalcularSalario se obtiene como:<br><br>salario(base) + calcularBono()<br><br>Dónde salario es el dato pasado por el constructor al momento de instanciar un objeto de este tipo, calcularBono() se obtiene mediante:<br><br>(actTerminadas * 0.1) * getSalario<br><br>actTerminadas siendo la cantidad de actividades que el programador ha hecho, dicha variable es modificable durante la ejecución del programa |

De esta forma, para el Programador se tiene que trabajar y hacer actividades si se quiere aumentar su sueldo, mientras que para el Gerente basta con modificar directamente su atributo bono.

### `Tablas de funciones`

Presentes en las carpetas de Javadocs[5]

## Pruebas

*Pruebas generadas para NO depender del usuario*

| | |
|---|---|
| **Ejercicio1** | ☑ 2 arreglos inicializados en elementos aleatorios deben ordenarse por QuickSort y Mergesort respectivamente |
| **Ejercicio2** | ☑ Demostrar el funcionamiento de los métodos creados para cada tipo de Empleado mediante impresiones de datos |

# Resultados

## Ejercicios en clase

| Ejercicio 0 | |
|---|---|
| Por BubbleSort | Por SelectionSort |

[5] Las carpetas llevan por nombre la clase que documentan

<table>
<tr>
<td>

```
Arreglo original
4
2
0
3
1
6
8

Arreglo ordenado
0
1
2
3
4
6
8
```

</td>
<td>

```
Arreglo original
0
1
2
3
4
6
8

Arreglo ordenado
0
1
2
3
4
6
8
```

</td>
</tr>
<tr>
<td colspan="2">Ejercicio 1</td>
</tr>
<tr>
<td colspan="2">

```
Mas adelante van a usar graficas
Area: 153.93804002589985
Un cilindro abstracto... ay que imaginacion
Area: 326.7256359733385
```

</td>
</tr>
</table>

## Ejercicios de tarea

### Ejercicio 1

Ordenamiento por ambos métodos

```
Arreglo original:
285 160 86 659 556 526 774 995 600 796
Arreglo ordenado por QuickSort:
86 160 285 526 556 600 659 774 796 995
Arreglo original:
734 710 196 547 582 259 411 48 375 530
Arreglo ordenado por MergeSort:
48 196 259 375 411 530 547 582 710 734
```

Se aprecia que para ambos arreglos generados aleatoriamente, ordena sus elementos

## Ejercicio 2

Creación de un empleado Gerente:

```
Detalles del Gerente 1:
+----
Nombre: Maria Macedo
Rol: Gerente
Salario Final: 25000.0
+----
```

Creación de un empleado Programador:

```
Detalles del Programador 1:
+----
Nombre: Daniel Gosling
Rol: Gerente
Salario Final: 5000.0
+----

Con conocimientos en:
Java
Python
C++

Actividades terminadas: 0
```

Note que el atributo tecnologías para dicho tipo de dato es un arreglo de Strings que guardan las tecnologías que sabe dicho programador, esto solo para fines de impresión

Programador 2 declarado:

```
Detalles del Programador 2:
+----
Nombre: Fernanda Lovelace
Rol: Gerente
Salario Final: 9000.0
+----

Con conocimientos en:
React
Angular
JavaScript

Actividades terminadas: 5
```

Aumento de salario para los **programadores**:

```
-- Los programadores se pueden esforzar y hacer actividades para aumentar su salario
El programador 1 ha terminado 4 actividades
+----
Nombre: Daniel Gosling
Rol: Gerente
Salario Final: 7000.0
+----

Con conocimientos en:
Java
Python
C++

Actividades terminadas: 4
```

```
El programador 2 ha terminado 3 actividades
+----
Nombre: Fernanda Lovelace
Rol: Gerente
Salario Final: 10800.0
+----

Con conocimientos en:
React
Angular
JavaScript

Actividades terminadas: 8
```

Se puede apreciar que para aumentar su salario, se tuvo que hacer que se incrementara el número de actividades terminadas, de tal forma que la fórmula hiciera efecto.

En cambio el Gerente solo aumenta su bono con los setters:

```
Actividades terminadas: 8
-- El gerente solo aumenta su bono
+----
Nombre: Maria Macedo
Rol: Gerente
Salario Final: 26000.0
+----
```

# Conclusiones

Diversos conceptos convergen en esta práctica, por un lado se tiene la herencia, que sigue desarrollandose como concepto clase en un paradigma orientado a objetos, ya que controla las relaciones que existen entre clases, favoreciendo la reutilización del código para así tener una superclase que defina los atributos y métodos de muchas otras subclases que la extiendan y ya sea implementen, modifiquen (sobreescriban) o concreten algunos de esos métodos.

Dentro de dicha herencia se puede emular la Múltiple con Interfaces, que son clases abstractas puras, que así como los archivos de cabecera en C, definen solo el prototipo (firma) de la función miembro, esperando que las implementaciones de dicha interfaz concreten y definan dichos métodos.

El Polimorfismo permite que una superclase pueda cambiar en tiempo de ejecución o de compilación, su forma; esto es su comportamiento de acuerdo al tipo de polimorfismo aplicado.

Se puede entender como aquel concepto que espera que el programa resuelva qué hacer al momento de detonar una llamada de acuerdo a la forma que tenga dicho objeto.

Un objeto puede tomar las formas de sus hijos (hacia abajo en la jerarquía de clases) pero no con sus hermanos (en el mismo nivel de herencia)[6]

Una clase abstracta espera que definas 1 o más métodos abstractos en su definición, esto para que en la herencia, alguna subclase que extienda dicha clase abstracta defina dicho método de acuerdo a sus necesidades. [7]

Dado que se cubrieron los requerimientos de los ejercicios y se comprendieron los conceptos vistos, el objetivo de la práctica fue logrado.

[6] Object es la única clase que no tiene superclase y es común para todos lo objetos en Java
[7] Las clases abstractas **no son instanciables**

# Apéndice A.

## Ordenamientos

### [3]

**BubbleSort**: Este algoritmo compara repetidamente elementos adyacentes de un arreglo y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que todo el arreglo está ordenado. BubbleSort tiene un rendimiento en el peor de los casos de $O(n^2)$ debido a la cantidad de comparaciones necesarias, ya que cada elemento debe ser comparado con todos los demás en múltiples pasadas

**SelectionSort**: SelectionSort selecciona el elemento mínimo (o máximo, según el orden) de una lista y lo intercambia con el primer elemento desordenado. Luego, este proceso se repite para el resto del arreglo hasta que esté completamente ordenado. Este algoritmo también tiene una complejidad de $O(n^2)$, ya que realiza nnn pasadas con nnn comparaciones en cada pasada para encontrar el mínimo

**QuickSort**: QuickSort utiliza el paradigma de divide y vencerás. Primero, selecciona un pivote y particiona el arreglo en dos subarreglos, uno con elementos menores al pivote y otro con elementos mayores. Luego aplica recursivamente el mismo proceso en cada subarreglo. QuickSort tiene un rendimiento promedio de $O(n\ log\ n)$, aunque en el peor de los casos puede ser $O(n^2)$ si los elementos están en orden desbalanceado

**MergeSort**: También basado en el principio de divide y vencerás, MergeSort divide el arreglo en mitades, ordena cada mitad recursivamente, y luego combina las mitades ordenadas para formar el arreglo final. Su complejidad en todos los casos es $O(n\ logn)$, ya que siempre divide el arreglo en mitades y luego fusiona los resultados de forma ordenad

# Referencias

[1] P. Deitel y H. Deitel. *CÓMO PROGRAMAR EN JAVA*. Sétima edición. México: Pearson Educación México, 2008. ISBN: 978-970-26-1190-5.

[2] "Programación Orientada a Objetos" notas de clase de 1323, División de Ingeniería Eléctrica, Facultad de Ingeniería de la Universidad Nacional Autónoma de México, Verano 2024.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, y C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009, pp. 1-1312. ISBN: 978-0-262-03384-8.