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

# Laboratorios de computación salas A y B

| | |
|---|---|
| *Profesora:* | Rocío Alejandra Aldeco Pérez |
| *Asignatura:* | Programación Orientada a Objetos |
| *Grupo:* | 6 |
| *No de Práctica(s):* | 7 y 8 |
| *Integrante(s):* | Ugalde Velasco Armando |
| *No. de Equipo de cómputo empleado:* | |
| *No. de Lista o Brigada:* | |
| *Semestre:* | 2021-1 |
| *Fecha de entrega:* | 27 de noviembre de 2020 |
| *Observaciones:* | |

CALIFICACIÓN: _____

**Práctica de Estudio 7 y 8: Herencia y polimorfismo**
Programación Orientada a Objetos Grupo 6
Facultad de Ingeniería
Departamento de Computación

**Objetivos de la práctica:**

- Implementar el concepto de polimorfismo en un lenguaje de programación orientado a objetos.
- Implementar el concepto de herencia en un lenguaje de programación orientado a objetos.

Realiza las siguientes actividades después de leer y revisar en clase la **Práctica de Estudio 7: Herencia** y la **Práctica de Estudio 8: Herencia**.

En esta práctica usarás dos funciones de ordenamiento ya existentes para implementar tus propias funciones sobre diferentes tipos de objetos. En el archivo **Sorting.java** (anexo a esta práctica) encontrarás las implementaciones de los algoritmos de ordenamiento por selección (*selection sort*) y por inserción (*insertion sort*) para ordenar arreglos con objetivos de tipo **Comparable** en orden ascendente.

**1.** El archivo **Numbers.java** lee un arreglo de enteros, llama al método **selectionSort**, ordena los elementos y luego imprime los números ordenados. Guarda **Sorting.java** y **Numbers.java** en tu directorio local. ¿Por qué es posible llamar al método **selectionSort** de la manera mostrada? Explica.

El método mencionado se encuentra definido de forma estática, es decir, es un miembro que pertenece a la clase o tipo como tal, no a las instancias. Por lo tanto, es posible llamarlo utilizando su nombre cualificado (*Sorting.selectionSort*), sin necesidad de crear una instancia de la clase.

**2.** Intenta compilar **Numbers.java**, no compilará. Revisa el error que te devuelve el compilador. El problema tiene que ver con la diferencia entre tipos de datos primitivos y objetos. Revisa el API de **Comparable**. Cambia el programa para que funcioné correctamente. *(OJO: no tienes que hacer muchos cambios ya que Java por sí solo se encarga de muchas conversiones)*. Explica que cambios realizaste y por qué. ¿Qué característica de la programación orientada a objetos se usa? Muestra tú código.

El arreglo que contiene los números a ordenar es de tipo **int[]**. Debido a que **int** no es una clase, sino un tipo primitivo, es imposible que implemente la interfaz **Comparable**, debido a su naturaleza. Dado que el método **selectionSort** requiere un argumento cuyo tipo sea un arreglo de objetos que implementen la interfaz **Comparable**, el llamarlo con un objeto de tipo **int[]** provoca un error de compilación, ya que, como se mencionó, éste no coincide con el tipo deseado. Para resolverlo, se declaró e inicializó el arreglo cambiando el tipo de sus elementos a **Integer**, es decir, la clase **wrapper** de **int**, la cual sí implementa la interfaz **Comparable**. En este caso, el concepto de **polimorfismo** se encuentra claramente presente, ya que es posible acceder a la funcionalidad de objetos de tipo **Integer** mediante la interfaz **Comparable**.

```
Integer[] intList;
intList = new Integer[size];
```

**Declaración e inicialización de arreglo con elementos de tipo Integer**

Es importante mencionar que, al momento de asignar los elementos respectivos al arreglo, se realiza un **autoboxing**, ya que el método **nextInt** retorna un dato de tipo **int**, pero la variable a la cual éste se asigna se encuentra declarada como **Integer**, por lo tanto, se realiza una conversión automática.

```
intList[i] = scan.nextInt();
```

**Autoboxing de int a Integer**

**3.** Escribe un programa llamado **Strings.java** similar a **Numbers.java**, que lea un arreglo de Strings y los ordene. Puedes copiar **Numbers.java** y editarlo. Explica que cambios realizaste y por qué. Muestra tú código.

Se cambió el tipo de los elementos del arreglo a **String**. Es decir, ahora el tipo de la lista que contendría los elementos a ordenar es **String[].**

```
String[] strList;
```

**Declaración de la lista de cadenas**

La cantidad de elementos a ordenar se solicitó de la misma forma, y se instanció el arreglo de cadenas con el tamaño indicado.

```
System.out.print("\n¿Cúantas cadenas quieres ordenar? ");
size = scan.nextInt();
strList = new String[size];
```

**Se solicita el número de elementos a ordenar y se instancia el arreglo**

Posteriormente, se solicitaron las cadenas de la misma forma que se solicitaron los enteros en **Numbers**, pero utilizando el método pertinente: **nextLine**. Nótese que, al inicio del ciclo, se ejecuta este método una vez para leer el carácter *"\n"* de la **stream** de la entrada estándar, ya que éste no se *"consumió"* al utilizar el método **nextInt**.

```
scan.nextLine();
for (int i = 0; i < size; i++)
{
    strList[i] = scan.nextLine();
}
```

**Se solicitan las cadenas a ordenar**

Finalmente, se ordenó e imprimió el arreglo de la misma forma que en **Numbers**:

```
Sorting.selectionSort(strList);
System.out.println("\nLas cadenas ordenadas son:");
for (int i = 0; i < size; i++)
{
    System.out.print(strList[i] + " ");
}
System.out.println();
```

**Se ordena e imprime el arreglo**

**4.** Modifica el método **insertionSort** para que ordene de forma descendente. Cambia **Strings.java** y **Numbers.java**, para que llamen a **insertionSort**. Corre ambos para verificar que el ordenamiento es correcto. Explica que cambios realizaste y por qué. Muestra tú código.

El método **compareTo** retorna un entero **negativo**, **cero**, o un entero **positivo** si el objeto en cuestión es **menor**, **igual** o **mayor** al objeto especificado en el argumento, respectivamente. En el algoritmo **insertionSort** se corrigen las violaciones de orden en el arreglo al iterar sobre éste, de tal forma que, antes de iterar en un elemento, todos los elementos que se encuentran antes de éste ya se encuentren ordenados, y únicamente sea necesario corregir las violaciones provocadas por el elemento actual al *"recorrerlo"* hacia la izquierda hasta que cumpla con la condición de orden respectiva.

Para ordenar el arreglo de forma descendente, es necesario cumplir con la condición de orden: $\mathbf{a_1 >= a_2 \ldots a_{n-1} >= a_n}$. Por lo tanto, al momento de corregir las violaciones provocadas por el elemento iterado, es necesario comprobar si éste es mayor a su elemento anterior. De ser así, será necesario recorrerlo para corregir la violación. Para lograr lo anterior, se utiliza el método **compareTo**, el cual, como se mencionó, retorna un entero positivo de presentarse esta situación. En otras palabras, se cambió la condición de **"menor a"** presente en el ciclo encargado de corregir las violaciones:

```
while (position > 0 && key.compareTo(list[position - 1]) > 0)
```

**Ciclo que corrige las violaciones provocadas por el elemento actual**

Finalmente, en los archivos **Strings** y **Numbers**, se sustituyó la utilización del método de ordenamiento anterior por **insertionSort**.

```
Sorting.insertionSort(intList);
Sorting.insertionSort(strList);
```

**Utilización del método insertionSort en las clases Strings y Numbers**

**5.** Ahora tienes la posibilidad de ordenar números y cadenas en orden ascendente y descendente. Crea un archivo **Main.java** que sea capaz de pasar los casos prueba en A**lphagrader** y haga uso de Strings.java y Numbers.java. En estos casos de prueba veras algunos con cadenas y otros con números. Para indicar que se ordenarán ascendentemente verás una A para descendentemente una D, esto al inicio de la lectura de datos.

Primero, se obtuvo la bandera que indicaría el tipo de orden a realizar y se declaró e inicializó una lista de elementos **Comparable**. Posteriormente, en caso de que el primer elemento fuera de tipo entero, se añadieron todos los números restantes utilizando el método **nextInt**. En caso contrario, se añadieron todos los elementos como cadenas. Finalmente, se convirtió la lista a un arreglo, el cual se ordenó e imprimió.

```
public static void main(String[] args)
{
    Scanner scanner = new Scanner(System.in);
    String flag = scanner.nextLine();
    List<Comparable> elements = new ArrayList<>();

    if (scanner.hasNextInt())
        while (scanner.hasNextInt()) elements.add(scanner.nextInt());
    else
        while (scanner.hasNextLine()) elements.add(scanner.nextLine());

    Comparable[] sorted = elements.toArray(new Comparable[0]);

    if (flag.equals("A")) Sorting.selectionSort(sorted);
    else Sorting.insertionSort(sorted);

    Arrays.stream(sorted).forEach(System.out::println);
}
```

**Método main**

**6.** Cuando estés seguro de que tu programa es correcto, súbelo a **Alphagrader**. Recuerda que si no pasas todos los test no obtendrás los puntos de ejecución.

**7.** Genera un documento **pdf** de esta práctica con la correspondiente carátula y las respuestas a las preguntas 1 a 4. Recuerda subir este documento en PDF a **Schoology** junto con el código de esta práctica *(los archivos .java).*




## CONCLUSIONES

La **herencia** y el **polimorfismo** son dos conceptos ubicuos en el paradigma orientado a objetos: el primero nos permite establecer relaciones jerárquicas entre clases, y el segundo permite que dos distintas clases presenten una misma interfaz, independientemente de su estructura interna.

Lo anterior, habilita una gran cantidad de posibilidades al momento de modelar y resolver problemas. Sin duda alguna, ambos conceptos se encuentran estrechamente relacionados, ya que, al existir una clase que hereda de otra, la clase hija automáticamente adquiere la misma interfaz pública de la clase padre, permitiendo que ésta sea accesada de la misma forma. En otras palabras, se presenta una forma de polimorfismo. Por otro lado, en la práctica también se presentó un ejemplo muy claro de la presencia del concepto anterior: se utilizó la interfaz **Comparable** para acceder a cierta funcionalidad común en instancias de distinto tipo, pero que la implementaran.