	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Dra. Rocío Alejandra Aldeco Pérez

Asignatura: Programación orientada a objetos -1323

Grupo: 6

No de Práctica(s): 7 - 8

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: 2023-1

Fecha de entrega: 4 de noviembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 7-8.

Herencia y polimorfismo.

OBJETIVOS

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

ACTIVIDADES

- Crear clases que implementen herencia.
- Generar una jerarquía de clases.
- A partir de una jerarquía de clases, implementar referencias que se comporten como diferentes objetos.

INTRODUCCIÓN

En la programación orientada a objetos, la **herencia** está en todos lados, de hecho, se podría decir que es casi imposible escribir el más pequeño de los programas sin utilizar **herencia**. Todas las clases que se crean dentro de la mayoría de los lenguajes de programación orientados a objetos heredan implícitamente de la clase *Object* y, por ende, se pueden comportar como objetos (que es la base del paradigma).

La herencia permite crear nuevos objetos que asumen las propiedades de objetos existentes. Una clase que es usada como base para heredarse es llamada **súper clase** o **clase base**. La clase derivada hereda todas las propiedades y métodos visibles de la clase base y, además, puede

agregar propiedades y métodos propios.

El término **polimorfismo** es constantemente referido como uno de los pilares de la programación orientada a objetos (junto con la Abstracción, el Encapsulamiento y la Herencia).

El término **polimorfismo** es una palabra de origen griego que significa **muchas formas**. En la programación orientada a objetos se refiere a la **propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos distintos**.

El **polimorfismo** consiste en conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases. Se puede aplicar tanto a métodos como a tipos de datos. Los métodos pueden evaluar y ser aplicados a diferentes tipos de datos de manera indistinta. Los tipos polimórficos son tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

El polimorfismo se puede clasificar en dos grandes grupos:

- Polimorfismo dinámico (o paramétrico): es aquel en el que **no se especifica el tipo de datos sobre el que se trabaja** y, por ende, se puede realizar todo tipo de datos compatible. Este tipo de polimorfismo también se conoce como programación genérica.
- Polimorfismo estático (o ad hoc): es aquel en el que **los tipos de datos que se pueden utilizar deben ser especificados** de manera explícita antes de ser utilizados.

INSTRUCCIONES

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

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**.
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 funcione correctamente. **Explica que cambios realizaste y por qué. ¿Qué característica de la programación orientada a objetos se usa? Muestra tu código**.
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 tu código**.
4. Modifica el método de *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 tu código**.
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 campos de prueba en Alphagrader y haga uso de *Strings.java* y *Numbers.java*. En estos casos de prueba verás 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.
6. Cuando estés seguro de que tu programa es correcto, súbelo a Alphagrader.

DESARROLLO

Descripción de actividades realizadas

Se realizó lo siguiente en una copia de los programas de *Sorting.java y Numbers.java* en un proyecto local:

1) Identificando en *numbers.java* el llamado a *selectionSort.*

```
Sorting.selectionSort(intList);
```

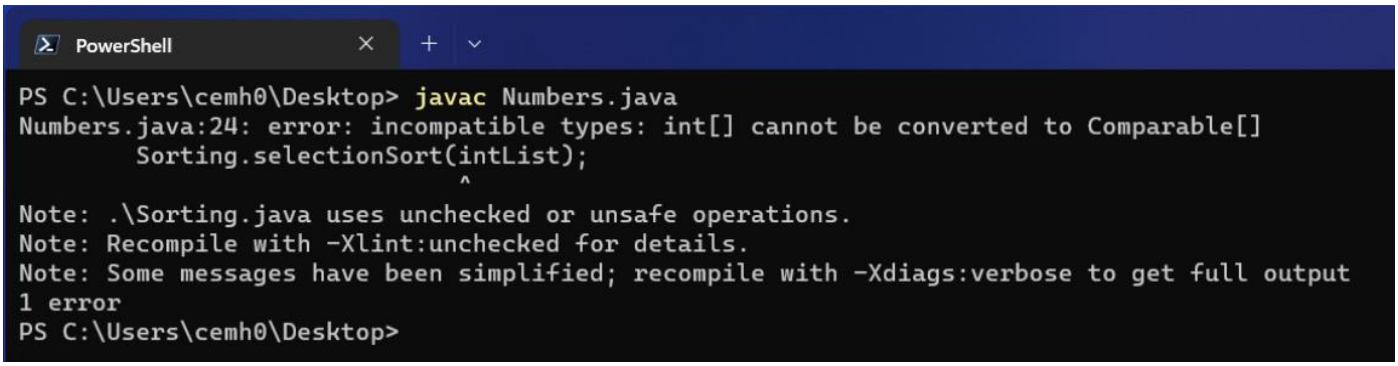
¿Por qué es posible llamar al método *selectionSort* de la manera mostrada? Explica.

Se observa que se llama al método con un arreglo de tipo *int[]* como parámetro, sin embargo, en su declaración, *selectionSort* recibe un parámetro de tipo *Comparable[]*.

Esto es posible gracias al polimorfismo de métodos. Así, un método puede recibir cualquier tipo de dato compatible como parámetro. Cuando el parámetro definido es una referencia a una clase, el método es capaz de recibir un objeto de este tipo o de cualquier subtipo de esa clase.

```
public static void insertionSort (Comparable[] list){
```

2) Compilando el archivo para encontrar el error



Como se vio en el punto anterior, el método de *insertionSort* recibe como parámetro un arreglo de tipo *Comparable* (en realidad, *Comparable* no puede ser visto como un objeto o tipo de dato dado que es una interfaz y no puede ser instanciada, solo implementada por otras clases).

Por otra parte, el parámetro que se envía al llamar al método es *intList*, declarada como un arreglo de datos tipo *int[]*, lo cual marca el error.

Explica que cambios realizaste y por qué. Muestra tu código.

Cambiando la declaración de *intList*. Dado que ahora *intList* es una instancia de la clase la clase *Integer* (no un tipo de dato primitivo), se debe modificar también la sentencia con *new*.

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

Este cambió se puede explicar al ver la API de Java:

compact1, compact2, compact3
java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedDateTime<D>, Delayed, Name, Path, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, AccessMode, AclEntryFlag, AclEntryPermission, AclEntryType, AddressingFeature.Responses, Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, CertPathValidatorException.BasicReason, Character, Character.UnicodeScript, CharBuffer, Charset, ChronoField, ChronoUnit, ClientInfoStatus, CollationKey, Collector.Characteristics, Component.BaselineResizeBehavior, CompositeName, CompoundName, CRLReason, CryptoPrimitive, Date, Date, DayOfWeek, Desktop.Action, Diagnostic.Kind, Dialog.ModalExclusionType, Dialog.ModalityType, DocumentationTool.Location, Double, DoubleBuffer, DropMode, Duration, ElementKind, ElementType, Enum, File, FileTime, FileVisitOption, FileVisitResult, Float, FloatBuffer, FormatStyle, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GraphicsDevice.WindowTranslucency, GregorianCalendar, GroupLayout.Alignment, HijrahChronology, HijrahDate, HijrahEra, Instant, IntBuffer, Integer,

Se encuentra a la clase *Integer* dentro de todas aquellas clases que implementan la interfaz *Comparable*. Esto implica que dentro de la clase *Integer* se implementan los métodos de *Comparable* permitiendo hacer comparaciones con instancias de la clase y, por lo tanto, haciendo posible el ordenamiento. En el caso anterior, con el tipo primitivo *int*, no es posible implementar un método. Por ello el cambio resulta ser cambiar de tipo primitivo *int* a una instancia de la clase *Integer*.

Sin embargo, incluso después de hecho lo anterior, aparece otra advertencia antes de compilar el programa:

Comparable is a raw type. References to generic type Comparable<T> should be parameterized.

Podemos volver a observar en la API que *T* en *Comparable* es un tipo genérico, y debe parametrizarse antes del tipo de retorno. Se pude parametrizar de la siguiente manera:

```
public static <T extends Comparable<? super T>> void insertionSort(T[] list) {
```

Este parámetro significa que el tipo *T* o alguna superclase de *T* deben implementar la interfaz *Comparable*. Por lo tanto, se modificó la firma de *insertionSort* y de *selectionSort* parametrizando antes del valor de retorno del método con *<T extends Comparable<? Super T>>*.

¿Qué característica de la programación orientada a objetos se usa?

La firma de estos métodos enseña como declarar un método genérico, haciendo uso del concepto de polimorfismo dinámico. Se dice que es polimorfismo dinámico dado que en estos métodos no se especifica el tipo de datos con los que se trabaja, pero se debe utilizar algún tipo de dato compatible.

3) En una copia del código de *Numbers.java* se modificó para trabajar con *Strings*.

Explica que cambios realizaste y por qué. Muestra tu código.

Las modificaciones que se hicieron en el código son, en esencia, adaptaciones para trabajar con *Strings*. El arreglo *stringList* (en lugar de *intList*) se declara como un arreglo de *Strings*, y sus elementos se leen del teclado con el método *scan.next()* (en lugar de *scan.nextInt()*). El llamado al método *insertionSort* es básicamente igual en sintaxis (pero se cambió el parámetro dado que se renombró el arreglo). Aquí es notorio el uso del polimrfismo, mensajes sintácticamente iguales para objetos de tipos distintos.

```
import java.util.Scanner;

public class Strings {
    // -----------------------------------------
    // Reads in an array of Strings, sorts them,
    // then prints them in sorted order.
    // -----------------------------------------
    public static void main(String[] args) {
        String[] stringList;
        int size;
        Scanner scan = new Scanner(System.in);

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

        System.out.println("\nDame las cadenas...");
        for (int i = 0; i < size; i++)
            stringList[i] = scan.next();
        Sorting.insertionSort(stringList);
        System.out.println("\nLas cadenas ordenadas son:");
        for (int i = 0; i < size; i++)
            System.out.print(stringList[i] + " ");
        System.out.println();
        scan.close();
    }
}
```

4) Se hizo el siguiente cambió a *insertionSort* para que realice ordenamientos descendentes

Explica que cambios realizaste y por qué. **Muestra tu código**.

```
public static <T extends Comparable<? super T>> void insertionSort(T[] list) {
        for (int index = 1; index < list.length; index++) {
            T key = list[index];
            int position = index;
            // Shift larger values to the left
            while (position > 0 && key.compareTo(list[position - 1]) > 0) {
                list[position] = list[position - 1];
                position--;
            }
            list[position] = key;
}
```

Originalmente, el algoritmo recorría los valores mayores hacia la derecha. Esto se realiza en el ciclo

```
while (position > 0 && key.compareTo(list[position - 1]) < 0) {
                list[position] = list[position - 1];
                position--;
}
list[position] = key;
```

Donde *key* es el iésimo valor que se está acomodando en el algoritmo y se compara con sus vecinos anteriores en el arreglo, es decir, con los números a la izquierda. Por otro lado, el método *compareTo* devuelve un número negativo si *key* es menor al número a su izquierda. Esto quiere decir que si para cualquier *key*, mientras los números a su izquierda sean mayores, la posición de *key* se recorre a la izquierda y cuando encuentra la posición correcta, se copia el valor de *key* allí.

La modificación es simple, si en orden ascendente los números mayores se recorren a la derecha, en orden ascendente los números mayores deben recorrerse a la izquierda, el cambio se refleja en

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

Donde, para cualquier *key*, mientras los números a su izquierda sean menores, la posición de *key* se recorre a la izquierda y cuando encuentra la posición correcta, se copia el valor de *key* allí.

5) Se creó la clase *Main* para pasar las pruebas del Alphagrader

Destacan los siguientes puntos:

La primera mitad del código se encarga de manejar los arreglos de enteros. Para saber si los valores de las entradas son números se utiliza la función *hasNextInt()*. Como se desconoce la cantidad de números (entradas) del programa (número necesario para definir el tamaño del arreglo donde se guardarán las entradas), se creó una cola encargada de almacenar temporalmente todas las entradas. Con esta cola es posible conocer la cantidad de entradas y crear el arreglo del tamaño exacto. Posteriormente, se desencolan los elementos para añadirlos al arreglo. Una vez, hecho esto y dependiendo del caso de la primera letra de la entrada (A o D) se hace el llamado a los métodos de ordenamiento y se imprime el arreglo ordenado.

```
if (stdin.hasNextInt()) {
        ArrayDeque<Integer> queue = new ArrayDeque<>();

        int temp = 0;
        while (stdin.hasNextInt()){
            temp = stdin.nextInt();
            queue.add(temp);
        }
        int size = queue.size();
        Integer arr[] = new Integer[size];
        for (int i = 0; i < size; i++) {
            arr[i] = queue.poll();
        }
        if (order == 'A')
            Sorting.selectionSort(arr);
        else
            Sorting.insertionSort(arr);
        for (int i = 0; i < size; i++)
            System.out.println(arr[i]);
}
```

La segunda mitad del código se encarga de los *Strings* y funciona de manera análoga a la primera parte.

```
else {
            ArrayDeque<String> queue = new ArrayDeque<>();
            String temp = "";
            while (stdin.hasNext()) {
                temp = stdin.next();
                queue.add(temp);
            }
            int size = queue.size();
            String[] arr = new String[size];
            for (int i = 0; i < size; i++) {
                arr[i] = queue.poll();
            }
            if (order == 'A')
                Sorting.selectionSort(arr);
            else
                Sorting.insertionSort(arr);
            for (int i = 0; i < size; i++)
                System.out.println(arr[i]);
        }
```

6) Para probar el funcionamiento del programa, se generó el archivo *zip* y se subío a Alphagrader.

Assignment: Practica 08 - Polimorfismo

Overview Tests Submissions

Submission

General Info	
Status	SUCCESS
Uploaded by	Emiliano Mendoza
Created at	November 02, 2022 06:08
Team members	Emiliano Mendoza
File size	2.41 KB
Download url	Download
Language	Java

CONCLUSIONES

En esta práctica se implementó el concepto de polimorfismo, más concretamente de método polimórfico al trabajar con un método que recibe un arreglo de un tipo de dato genérico y que puede hacer ordenamientos de tipos de datos compatibles con la interfaz *Comparable*.

A partir de la jerarquía de clases e interfaces, se encontró que algunas clases como *Integer*, *String*, *Float*, *Double*, etc. implementan la interfaz *Comparable*. Gracias a ello fue posible implementar una referencia, *T*, que se compartara como diferentes objetos.

Además, fue posible identificar el tipo de polimorfismo aplicado a este método, que es el de polimorfismo dinámico o también conocido como programación genérica. Con este tipo de polimorfismo, no es necesario especificar el tipo de datos sobre el que se trabaja, permitiendo la utilización de cualquier tipo de datos compatible.

Dado que se comprendieron y se implementaron los conceptos de polimorfismo y herencia, se concluye que se cumplieron los objetivos de la práctica.

REFERENCIAS

- *Barnes David, Kölling Michael*
  Programación Orientada a Objetos con Java.
  Tercera Edición.
  Madrid
  Pearson Educación, 2007

- *Deitel Paul, Deitel Harvey*
  Cómo programar en Java.
  Séptima Edición.
  México
  Pearson Educación, 2008

- *Martín, Antonio*
  Programador Certificado Java 2.
  Segunda Edición.
  México
  Alfaomega Grupo Editor, 2008

- *Dean John, Dean Raymond*
  Introducción a la programación con Java.
  Primera Edición.
  México
  Mc Graw Hill, 2009

Yo, Carlos Emiliano Mendoza Hernández, hago mención que esta práctica fue de mi autoría.