Acosta Porcayo Alan Omar, Gutiérrez Grimaldo Alejandro, Medina Villa Samuel

Práctica 8. Polimorfismo

Resumen

En esta práctica se implementa el concepto de polimorfismo a través de una jerarquía de clases. Se crean referencias que pueden comportarse como diferentes objetos dentro de esta jerarquía. Se diseñan métodos polimórficos que permiten a las referencias acceder a comportamientos específicos de cada subclase. Esto demuestra la flexibilidad y extensibilidad que el polimorfismo aporta al código, permitiendo la interacción uniforme con objetos diversos. En resumen, la práctica destaca la importancia del polimorfismo como una herramienta clave en la programación orientada a objetos para mejorar la modularidad y escalabilidad del software.

Introducción

Polimorfismo

El polimorfismo es un pilar esencial de la programación orientada a objetos. Su nombre proviene del griego y significa "muchas formas". En este contexto, se refiere a la capacidad de enviar mensajes sintácticamente iguales a objetos de tipos distintos. Esto implica que un objeto de una clase puede comportarse como un objeto de cualquiera de sus subclases.

El polimorfismo se puede aplicar tanto a métodos como a tipos de datos. Los métodos pueden evaluarse y aplicarse 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. Existen dos clasificaciones principales del polimorfismo:

- **Polimorfismo dinámico:** En este enfoque, no se especifica el tipo de datos sobre el que se trabaja, lo que permite recibir y utilizar todo tipo de datos compatibles. También se conoce como programación genérica.
- **Polimorfismo estático:** En este caso, se deben especificar de manera explícita los tipos de datos que se pueden utilizar antes de utilizarlos.

Clases Abstractas

Las clases abstractas son modelos que no pueden crear objetos debido a que definen la existencia de métodos, pero no su implementación. Algunas características de las clases abstractas incluyen:

- Pueden contener métodos abstractos (sin implementación) y métodos concretos.
- Pueden contener atributos.
- Pueden heredar de otras clases.
- Para declarar una clase abstracta en Java, se utiliza la palabra reservada "*abstract*" antes de la palabra "*class*".

Interfaces

Las interfaces son clases abstractas puras, lo que significa que todos los métodos en una interfaz son abstractos (sin implementación). Algunas características de las interfaces incluyen:

- Todos los métodos son siempre públicos y abstractos.
- Pueden contener atributos que son públicos, estáticos y finales.
- Las clases que implementan una interfaz deben definir (implementar) los métodos declarados en la interfaz.
- Las interfaces pueden heredar de otras interfaces y permiten la herencia múltiple.

Implementación Múltiple

Una clase puede implementar varias interfaces, y se requiere que la clase implemente todos los métodos de

Acosta Porcayo Alan Omar Ing. en Computación 320206102
Gutiérrez Grimaldo Alejandro Ing. en Computación 320282098
Medina Villa Samuel Ing. en Computación 320249538

las interfaces que se incluyan.

Herencia Múltiple entre Interfaces

Las interfaces pueden heredar de otras interfaces, permitiendo la herencia múltiple de interfaces. La clase que implementa una interfaz que hereda de otras interfaces debe definir los métodos de todas las interfaces.

Atributos en las Interfaces

En las interfaces, todos los atributos son públicos, estáticos y finales. Estos atributos se utilizan comúnmente para definir constantes que pueden ser accedidas sin crear objetos.

Esta información proporciona una comprensión sólida de los conceptos de polimorfismo, clases abstractas e interfaces en la programación orientada a objetos, lo que es fundamental para diseñar sistemas de software flexibles y extensibles.

Objetivos

- Implementar el concepto de polimorfismo en un lenguaje de programación orientado a objetos.
- Comprender y diferenciar entre el polimorfismo dinámico y estático, y demostrar la capacidad de aplicar ambos tipos de polimorfismo en situaciones específicas.
- Diseñar interfaces que definen comportamientos específicos y permiten a las clases implementar múltiples interfaces para lograr una mayor flexibilidad en la interacción entre objetos.

Metodología

Ejemplo de Polimorfismo

Interfaz *LifeForm*

```
public interface LifeForm {
  void eat(int quantity);

  int getHunger();

  int getYear();

  String move();
  String breathe();
  String reproduce();
}
```

Clase abstracta *AbstractAnimal*

```
public abstract class AbstractAnimal implements LifeForm{
  static private int maxId = 0;
  private int hunger;
  private String name;
  private int id;
  private int year;

  AbstractAnimal(String name, int year, int hunger) {
    maxId++;
    this.id = maxId;
    this.hunger = hunger;
    this.year = year;
    this.name = name;
  }

  AbstractAnimal(String name, int year) {
    this(name, year, 0);
  }

  public void eat(int quantity) {
    hunger -= quantity;
    hunger = hunger < 0 ? 0 : hunger;
  }

  public String getName() {
    return name;
  }

  public int getHunger() {
    return hunger;
  }

  public int getId() {
    return id;
  }

  public int getYear() {
    return year;
  }

  abstract public String move();
  abstract public String breathe();
  abstract public String reproduce();
}
```

Clase *Mammal*

```
public class Mammal extends AbstractAnimal {
  Mammal(String name, int year) {
    super(name, year);
  }

  public String move() {
    return "walk";
  }

  public String breathe() {
    return "lungs";
  }

  public String reproduce() {
    return "live births";
  }
}
```

Clase *Bird*

```
public class Bird extends AbstractAnimal {
  Bird (String name, int year) {
    super(name, year, 3);
  }

  public String move() {
    return "fly";
  }

  public String breathe() {
    return "lungs";
  }

  public String reproduce() {
    return "eggs";
  }
}
```

Clase *Fish*

```
public class Fish extends AbstractAnimal {
  Fish (String name, int year) {
    super(name, year);
  }

  public String move() {
    return "swim";
  }

  public String breathe() {
    return "gills";
  }

  public String reproduce() {
    return "eggs";
  }
}
```

Main

```
import java.util.ArrayList;
import java.util.List;

public class Main {
  @SuppressWarnings("ComparatorCombinators")
  public static void main(String[] args) {
    List<AbstractAnimal> animals = new ArrayList<>();
    animals.add(new Mammal("Panda", 1869));
    animals.add(new Mammal("Zebra",1778));
    animals.add(new Mammal("Koala",1816));
    animals.add(new Mammal("Sloth",1804));
    animals.add(new Mammal("Armadillo",1758));
    animals.add(new Mammal("Raccoon",1758));
    animals.add(new Mammal("Bigfoot",2021));

    animals.add(new Bird("Pigeon", 1837));
    animals.add(new Bird("Peacock",1821));
    animals.add(new Bird("Toucan",1758));
    animals.add(new Bird("Parrot",1824));
    animals.add(new Bird("Swan",1758));

    animals.add(new Fish("Salmon", 1758));
    animals.add(new Fish("Catfish", 1817));
    animals.add(new Fish("Perch", 1758));

    animals.sort((animal1, animal2) -> animal2.getYear() - animal1.getYear());
    animals.forEach(animal -> {
      System.out.println(animal.getName() + " " + animal.getYear());
    });

    System.out.println("\n-------Alphabetical-----------");
    animals.sort((animal1, animal2) -> animal1.getName().compareTo(animal2.getName()));
    animals.forEach(animal -> {
      System.out.println(animal.getName());
    });

    System.out.println("\n-------Movement-----------");
    animals.sort((animal1, animal2) -> animal1.move().compareTo(animal2.move()));
    animals.forEach(animal -> {
      System.out.println(animal.getName() + " " + animal.move());
    });

    System.out.println("\n-------Breathers-----------");
    animals.forEach(animal -> {
      if (animal.breathe().equals("lungs")) {
        System.out.println(animal.getName() + " " + animal.breathe());
      }
    });

    System.out.println("\n-------Breathers in 1758-----------");
    animals.forEach(animal -> {
      if (animal.breathe().equals("lungs") && animal.getYear() == 1758) {
        System.out.println(animal.getName() + " " + animal.breathe());
      }
    });
  }
}
```

Ejecución

```
Bigfoot 2021
Panda 1869
Pigeon 1837
Parrot 1824
Peacock 1821
Catfish 1817
Koala 1816
Sloth 1804
Zebra 1778
Armadillo 1758
Raccoon 1758
Toucan 1758
Swan 1758
Salmon 1758
Perch 1758

-------Alphabetical-----------
Armadillo
Bigfoot
Catfish
Koala
Panda
Parrot
Peacock
Perch
Pigeon
Raccoon
Salmon
Sloth
Swan
Toucan
Zebra

-------Movement-----------
Parrot fly
Peacock fly
Pigeon fly
Swan fly
Toucan fly
Catfish swim
Perch swim
Salmon swim
Armadillo walk
Bigfoot walk
Koala walk
Panda walk
Raccoon walk
Sloth walk
Zebra walk

-------Breathers-----------
Parrot lungs
Peacock lungs
Pigeon lungs
Swan lungs
Toucan lungs
Armadillo lungs
Bigfoot lungs
Koala lungs
Panda lungs
Raccoon lungs
Sloth lungs
Zebra lungs

-------Breathers in 1758-----------
Swan lungs
Toucan lungs
Armadillo lungs
Raccoon lungs
```

Resultados

Problema 1

Existen otros lenguajes de programación orientados objetos tales como:

a. KOTLIN (JetBrains /2016)

b. DART (Google / 2011)

c. Groovy (JCP /2003)

d. C# (Microsoft / 2000)

e. Python (Rossum / 1991)

Escoja un lenguaje de programación de la lista y estudie como se presenta el polimorfismo (estático y dinámico).

Problema 2

Traduzca el código del Reino Animal visto en clase al lenguaje seleccionado en el punto anterior y conteste las siguientes preguntas en conclusiones:

a. ¿Qué tan transparente fue realizar dicha traducción? La traducción del código de Java a Dart fue relativamente transparente en términos de lógica y estructura. La mayor parte de la lógica se mantuvo, pero hubo ajustes necesarios debido a diferencias en sintaxis, tipos de datos y manejo de nulos entre los dos lenguajes. En general, la similitud en la orientación a objetos facilitó la traducción, pero se requirió una adaptación para que el código funcionara en el nuevo entorno.

   Mientras que la traducción del código de Java a Kotlin fue altamente transparente y fluida. Kotlin está diseñado para ser compatible con Java, lo que hace que la mayoría de la lógica y estructura del código se traduzcan de manera directa. Kotlin ofrece una sintaxis más concisa y características adicionales que pueden mejorar la legibilidad y el mantenimiento del código.

b. ¿El concepto de polimorfismo se aplica de la misma manera? El concepto de polimorfismo se aplica de manera similar en Java, Dart y Kotlin, ya que todos son lenguajes orientados a objetos que admiten herencia y anulación de métodos. Sin embargo, cada lenguaje tiene su propia sintaxis y características específicas para lograr el polimorfismo. En Java, se utiliza la palabra clave “extends” para la herencia y “@Override” para anular métodos. En Dart, se hereda de manera implícita y se pueden anular métodos por defecto. En Kotlin, se utiliza “override” para anular métodos y se pueden usar “clases selladas” para limitar las subclases. A pesar de las diferencias en la implementación, el concepto subyacente de polimorfismo es una característica común en estos lenguajes orientados a objetos.

c. ¿Se consideraría Java como el padre de estos lenguajes? Java no es el “padre” de Kotlin y Dart en el sentido de herencia de lenguajes, pero ha influido en su desarrollo. Kotlin se diseñó como una extensión de Java con interoperabilidad completa, mientras que Dart comparte similitudes en programación orientada a objetos. Estos lenguajes se inspiraron en Java y buscan mejorar sus capacidades en áreas específicas. Java influyó en su evolución, pero no son descendientes directos.

Dart

Clase *LifeForm*

```dart
abstract class LifeForm {
  void eat(int quantity);

  int getHunger();

  int getYear();

  String move();
  String breathe();
  String reproduce();
}
```

Clase *AbstractAnimal*

```dart
import 'LifeForm.dart';

abstract class AbstractAnimal implements LifeForm {
  static int maxId = 0;
  late int hunger;
  late String name;
  late int id;
  late int year;

  AbstractAnimal(String name, int year, int hunger) {
    this.hunger = hunger;
    this.name = name;
    this.year = year;
    maxId++;
    id = maxId;
  }

  AbstractAnimal.namedYear(String name, int year) : this(name, year, 0);

  void eat(int quantity) {
    hunger -= quantity;
    hunger = hunger < 0 ? 0 : hunger;
  }

  String getName() {
    return name;
  }

  int getHunger() {
    return hunger;
  }

  int getId() {
    return id;
  }
```

```
  int getYear() {
    return year;
  }

  String move();
  String breathe();
  String reproduce();
}
```

Clase *Mammal*

```
import 'AbstractAnimal.dart';

class Mammal extends AbstractAnimal {
  Mammal(String name, int year) : super.namedYear(name, year);

  @override
  String move() {
    return "walk";
  }

  @override
  String breathe() {
    return "lungs";
  }

  @override
  String reproduce() {
    return "live births";
  }
}
```

Clase *Bird*

```
import 'AbstractAnimal.dart';

class Bird extends AbstractAnimal {
  Bird(String name, int year) : super(name, year, 3);

  @override
  String move() {
    return "fly";
  }

  @override
  String breathe() {
    return "lungs";
  }

  @override
  String reproduce() {
    return "eggs";
  }
}
```

Clase *Fish*

```
import 'AbstractAnimal.dart';

class Fish extends AbstractAnimal {
  Fish(String name, int year) : super.namedYear(name, year);

  @override
  String move() {
    return "swim";
  }

  @override
  String breathe() {
    return "gills";
  }

  @override
  String reproduce() {
    return "eggs";
  }
}
```

Main

```
import 'AbstractAnimal.dart';
import 'Bird.dart';
import 'Fish.dart';
import 'Mammal.dart';

void main() {
  List<AbstractAnimal> animals = [];
  animals.add(Mammal("Panda", 1869));
  animals.add(Mammal("Zebra", 1778));
  animals.add(Mammal("Koala", 1816));
  animals.add(Mammal("Sloth", 1804));
  animals.add(Mammal("Armadillo", 1758));
  animals.add(Mammal("Raccoon", 1758));
  animals.add(Mammal("Bigfoot", 2021));

  animals.add(Bird("Pigeon", 1837));
  animals.add(Bird("Peacock", 1821));
  animals.add(Bird("Toucan", 1758));
  animals.add(Bird("Parrot", 1824));
  animals.add(Bird("Swan", 1758));

  animals.add(Fish("Salmon", 1758));
  animals.add(Fish("Catfish", 1817));
  animals.add(Fish("Perch", 1758));

  animals.sort((animal1, animal2) => animal2.getYear() - animal1.getYear());
  animals.forEach((animal) {
    print('${animal.getName()} ${animal.getYear()}');
  });
  print("\n-------Alphabetical-----------");
  animals.sort(
      (animal1, animal2) => animal1.getName().compareTo(animal2.getName()));
  animals.forEach((animal) {
    print('${animal.getName()}');
  });

  print("\n-------Movement-----------");
  animals.sort((animal1, animal2) => animal1.move().compareTo(animal2.move()));
  animals.forEach((animal) {
    print('${animal.getName()} ${animal.move()}');
  });

  print("\n-------Breathers-----------");
  animals.forEach((animal) {
    if (animal.breathe() == "lungs") {
      print('${animal.getName()} ${animal.breathe()}');
    }
  });

  print("\n-------Breathers in 1758-----------");
  animals.forEach((animal) {
    if (animal.breathe() == "lungs" && animal.getYear() == 1758) {
      print('${animal.getName()} ${animal.breathe()}');
    }
  });
}
```

Kotlin

Clase *LifeForm*

```
interface LifeForm {
  fun eat(quantity: Int)
  fun getHunger(): Int
  fun getYear(): Int
  fun move(): String
  fun breathe(): String
  fun reproduce(): String
}
```

Clase *AbstractAnimal*

```
import LifeForm

abstract class AbstractAnimal : LifeForm {
  companion object {
    private var maxId = 0
  }

  private var hunger = 0
  private val name: String
  private val id: Int
  private val year: Int

  constructor(name: String, year: Int, hunger: Int) {
    maxId++
    this.id = maxId
    this.hunger = hunger
    this.year = year
    this.name = name
  }

  constructor(name: String, year: Int) : this(name, year, 0)
```

```
  override fun eat(quantity: Int) {
    hunger -= quantity
    hunger = if (hunger < 0) 0 else hunger
  }

  override fun getHunger(): Int {
    return hunger
  }

  override fun getYear(): Int {
    return year
  }

  override fun getName(): String {
    return name
  }

  abstract override fun move(): String
  abstract override fun breathe(): String
  abstract override fun reproduce(): String
}
```

Clase *Mammal*

```
import AbstractAnimal

class Mammal(name: String, year: Int) : AbstractAnimal(name, year) {
  override fun move(): String {
    return "walk"
  }

  override fun breathe(): String {
    return "lungs"
  }

  override fun reproduce(): String {
    return "live births"
  }

  override fun getName(): String {
    return super.getName()
  }
}
```

Clase *Bird*

```
import AbstractAnimal

class Bird(name: String, year: Int) : AbstractAnimal(name, year, 3) {
  override fun move(): String {
    return "fly"
  }

  override fun breathe(): String {
    return "lungs"
  }

  override fun reproduce(): String {
    return "eggs"
  }

  override fun getName(): String {
    return super.getName()
  }
}
```

Clase *Fish*

```
import AbstractAnimal

class Fish(name: String, year: Int) : AbstractAnimal(name, year) {
  override fun move(): String {
    return "swim"
  }

  override fun breathe(): String {
    return "gills"
  }

  override fun reproduce(): String {
    return "eggs"
  }

  override fun getName(): String {
    return super.getName()
  }
}
```

Main

```
import Mammal
import Bird
import Fish
import AbstractAnimal
import LifeForm

fun main() {
  val animals: MutableList<AbstractAnimal> = mutableListOf()

  animals.add(Mammal("Panda", 1869))
  animals.add(Mammal("Zebra", 1778))
  animals.add(Mammal("Koala", 1816))
  animals.add(Mammal("Sloth", 1804))
  animals.add(Mammal("Armadillo", 1758))
  animals.add(Mammal("Raccoon", 1758))
  animals.add(Mammal("Bigfoot", 2021))

  animals.add(Bird("Pigeon", 1837))
  animals.add(Bird("Peacock", 1821))
  animals.add(Bird("Toucan", 1758))
  animals.add(Bird("Parrot", 1824))
  animals.add(Bird("Swan", 1758))

  animals.add(Fish("Salmon", 1758))
  animals.add(Fish("Catfish", 1817))
  animals.add(Fish("Perch", 1758))

  animals.sortByDescending { it.getYear() }
  animals.forEach { animal ->
    println("${animal.getName()} ${animal.getYear()}")
  }

  println("-------Alphabetical-----------")
  animals.sortBy { it.getName() }
  animals.forEach { animal ->
    println(animal.getName())
  }

  println("-------Movement-----------")
  animals.sortBy { it.move() }
  animals.forEach { animal ->
    println("${animal.getName()} ${animal.move()}")
  }

  println("-------Breathers-----------")
  animals.filter { it.breathe() == "lungs" }.forEach { animal ->
    println("${animal.getName()} ${animal.breathe()}")
  }

  println("-------Breathers in 1758-----------")
  animals.filter { it.breathe() == "lungs" && it.getYear() == 1758 }.forEach { animal ->
    println("${animal.getName()} ${animal.breathe()}")
  }
}
```

Problema 3

Realice una tabla comparativa de desventajas y ventajas de la sobreescritura y sobrecarga de métodos y concluya desde su punto de vista cuál de las dos maneras de polimorfismo es más útil y porqué.

Sobreescritura

Ventajas	Desventajas
La capacidad de sobrescribir habilita el polimorfismo, lo que implica que es posible invocar tanto el método de la superclase como el de la subclase de forma consistente, simplificando así la creación de código genérico.	La sobreescritura se encuentra ligada a la herencia, lo que puede dar lugar a una estrecha interconexión entre las clases, lo cual no siempre es la elección ideal en el diseño de software.
Simplifica la tarea de mantener el código, puesto que posibilita la modificación de la forma en que se ejecuta un método en una clase hija sin que ello tenga repercusiones en las demás secciones del código que hacen uso de la clase base.	Se necesita mantener las firmas de los métodos idénticas, lo que implica que no tienes la libertad de alterar la cantidad o tipo de parámetros, ni el tipo de valor de retorno.
La capacidad de sobreescritura permite que las subclases sustituyan la funcionalidad heredada de la superclase, lo que resulta esencial para adaptar el comportamiento de las clases según las necesidades específicas.	

Sobrecarga

Ventajas	Desventajas
Habilita la creación de métodos que comparten el mismo nombre, pero poseen diferentes conjuntos de parámetros, lo que simplifica la creación de métodos con funcionalidades afines.	En situaciones particulares, la sobrecarga puede generar confusión si los argumentos no son lo bastante diferentes entre sí, lo que podría dar lugar a errores de compilación.
Los nombres de los métodos sobrecargados pueden ser formulados de manera más intuitiva y descriptiva por sí mismos, lo que hace que el código sea más comprensible y fácil de emplear.	El exceso de sobrecarga puede dar lugar a un código enredado y de complicada comprensión, especialmente cuando las diferencias entre las versiones sobrecargadas no son evidentes.
No se encuentra restringido por las mismas firmas de método que en la sobreescritura, lo que brinda una mayor libertad y flexibilidad en la definición de los métodos.	

La decisión entre utilizar la sobreescritura o la sobrecarga se basa en los requisitos de tu programa y la organización de tus clases. Ambos tipos de polimorfismo tienen su utilidad y se aplican en contextos diversos. En general, la sobreescritura es más útil cuando se trabaja con clases relacionadas entre sí, mientras que la sobrecarga es más útil cuando se trabaja con clases que no están relacionadas entre sí.

Problema 4

Implemente la solución del siguiente problema: https://edabit.com/challenge/6RStzK9uub9vHDt53

Pero considere la programación de los métodos polimórficos:

a. *sortByLength()* ascendente y descendente

b. *sortByAlphabet()* de a-z y z-a

Solución

```
import java.util.ArrayList;

public class Problema4{
  public class Sort{
    private static ArrayList<String> split(String s) {
      String[] words = s.split(" ");
      ArrayList<String> wordsAL = new ArrayList<>();
      for (int i = 0; i < words.length; i++)
        wordsAL.add(words[i]);
      return wordsAL;
    }

    private static String append(ArrayList<String> words){
        String result = "";
        for (int i = 0; i < words.size(); i++)
          result += words.get(i) + " ";
        return result;
    }

    public static String sortByLength(String s){
        ArrayList<String> words = new ArrayList<>();
        words = split(s);
        words.sort((word1, word2) -> word1.length() - word2.length());
        return append(words);
    }

    public static String sortByLength(String s, String a){
        ArrayList<String> words = new ArrayList<>();
        words = split(s);
        words.sort((word1, word2) -> word2.length() - word1.length());
        return append(words);
    }

    public static String sortByAlphabet(String s){
        ArrayList<String> words = new ArrayList<>();
        words = split(s);
        words.sort((word1, word2) -> word1.toLowerCase().charAt(0)
            - word2.toLowerCase().charAt(0));
        return append(words);
    }

    public static String sortByAlphabet(String s, String a){
        ArrayList<String> words = new ArrayList<>();
        words = split(s);
        words.sort((word1, word2) -> word2.toLowerCase().charAt(0)
            - word1.toLowerCase().charAt(0));
        return append(words);
    }
  }

  public static void main(String[] args) {
    String s = "Hello World in Java";

    System.out.println("String: " + s);

    System.out.println("\n- Ordenacion por longitud -");
    System.out.println("Ascendente: " + Sort.sortByLength(s));
    System.out.println("Descendente: " + Sort.sortByLength(s, "s"));

    System.out.println("\n- Ordenacion alfabetica -");
    System.out.println("a-z: " + Sort.sortByAlphabet(s));
    System.out.println("z-a: " + Sort.sortByAlphabet(s, "s"));
```

```
  }
}
```

Ejecución

```
String: Hello World in Java

- Ordenación por longitud -
Ascendente: in Java Hello World
Descendente: Hello World Java in

- Ordenación alfabética -
a-z: Hello in Java World
z-a: World Java in Hello
```

Conclusiones

El polimorfismo es un concepto fundamental en la programación orientada a objetos que permite que objetos de diferentes tipos puedan interactuar de manera uniforme y eficiente. A lo largo de la información proporcionada, hemos explorado en profundidad el polimorfismo, sus tipos (dinámico y estático) y su aplicación en diferentes contextos

El polimorfismo proporciona flexibilidad y extensibilidad al código, lo que es esencial para el diseño de sistemas de software versátiles y adaptables.

Entender y aplicar de manera efectiva el polimorfismo es esencial para nosotros los programadores ya que contribuye a la construcción de software de alta calidad.

Referencias

Solano, J. (2017, 20 enero). *Manual de prácticas de Programación Orientada a Objetos*. Laboratorio de Computación Salas A y B. http://lcp02.fi-b.unam.mx/

Zoark. *Sort by Length*. Edabit. https://edabit.com/challenge/6RStzK9uub9vHDt53