PRACTICO 4

Clases Concretas, Herencia y Polimorfismo Dinámico

Ej. 4.1

Ejecutar la siguiente porción de código:

```
class Animal {
 String nombre;
 Animal(String nombre) {
   this.nombre = nombre;
 }
 void hacerSonido() {
   System.out.println("El animal hace un sonido.");
 }
}

class Perro extends Animal {
 Perro(String nombre) {
   super(nombre);
 }
 @Override
 void hacerSonido() {
   super.hacerSonido();
   System.out.println(nombre + " dice: ¡Guau!");
 }
}

class Gato extends Animal {
 Gato(String nombre) {
   super(nombre);
 }
 @Override
 void hacerSonido() {
   System.out.println(nombre + " dice: ¡Miau!");
 }
}

class Medusa extends Animal {
 Medusa(String nombre) {
   super(nombre);
 }
 @Override
 void hacerSonido() {
   // la medusa no emite sonidos
 }
}
```

```
public class EjemploSuper {
 public static void main(String[] args) {
   Animal animalito = new Animal("Raul");
   Animal perro = new Perro("Firulais");
   Animal gato = new Gato("Misu");
   Animal medusa = new Medusa("Lola");

   animalito.hacerSonido();
   perro.hacerSonido(); // Firulais dice: ¡Guau!
   gato.hacerSonido();  // Misu dice: ¡Miau!
   medusa.hacerSonido();  // Lola  no dice nada
 }
}
```

¿Por qué se comportan de maneras distintas cada uno de las instancias?

Ej. 4.2

Codificar en Java la clase **PruebaIncompleta** que posee los siguientes métodos:

- `public void texto1(){ System.out.println(“Esta es una”); }`
- `public void texto2(){ System.out.print(“prueba”); }`
- `public void texto3(){ System.out.println(“incompleta”); }`
- `public void texto4(){ System.out.println(“en Java”); }`
- `public final void mensaje(){ texto1(); texto2(); texto3(); texto4(); }`

Codificar luego una clase *Prueba* que herede de *PruebaIncompleta*, redefiniendo apropiadamente y reusando lo máximo posible de los métodos heredados, para que la salida del método *mensaje()* en Prueba sea exactamente la siguiente:

```
Esta es una
prueba de
redefinición de métodos
```

Ej. 4.3

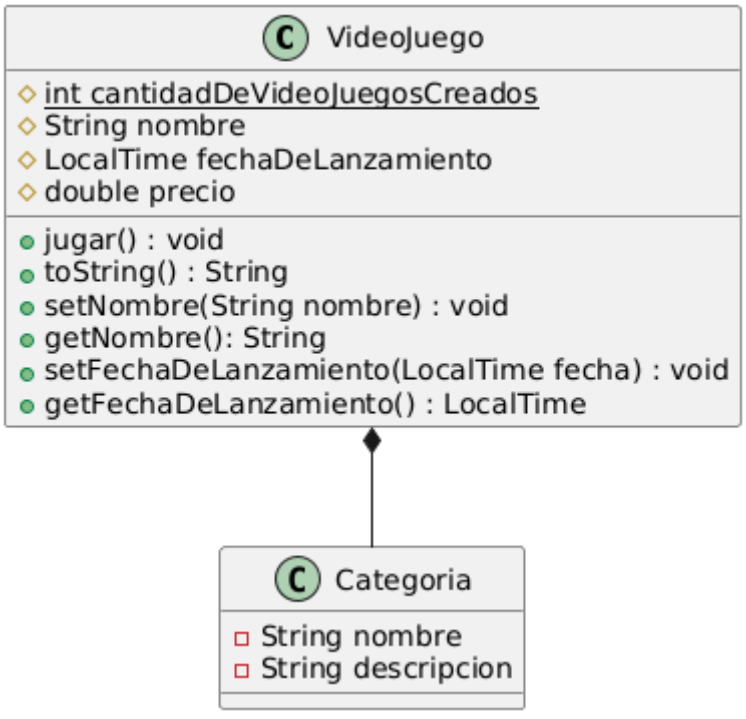
Compile, ejecute y analice el siguiente código. ¿Cuál es la diferencia entre el constructor de esta clase y los constructores de los ejercicios anteriores? ¿Cuál es la finalidad de que el constructor tenga esta característica? Completar el mensaje dentro del método getInstancia.

```
public class Misterio {
 private String nombre;
 private static Misterio misterioso;

 private Misterio(String nombre) {
   this.nombre = nombre;
   System.out.println("Mi nombre es: " + this.nombre);
 }
```

```
public static Misterio getInstancia(String nombre) {
  if (misterioso == null){
    misterioso = new Misterio(nombre);
  }
  else{
    System.out.println("...");
  }

  return misterioso;
}

public String getNombre(){
  return nombre;
}

public static void main(String[] args) {

    Misterio chavito = Misterio.getInstancia("El Chavo del
Ocho");
  Misterio ramon = Misterio.getInstancia("Don Ramon");
  System.out.println(ramon.getNombre());
  System.out.println(chavito.getNombre());
}

}
```

Ej. 4.4

1. Escribir en Java las siguientes clases:



2. Definir una clase **Catálogo** que contenga **VideoJuegos** (utilizando objetos del punto anterior). Además, debe ser posible **agregar/eliminar/listar** los **video juegos** del usuario. Implementar estas acciones utilizando arreglos, Vectores, ArrayList. **Analizar semejanzas y diferencias de un arreglo con respecto a la clase Vector, como también con ArrayList**.
   Utilizar la clase Scanner de la JDK, para lograr la funcionalidad solicitada.

Ej. 4.5

Crear una clase ***Mazo*** que almacene, en un ***array* o *vector***, un conjunto de ***Naipes*** para los cuales se debe conocer su número y su palo, y permita agregar nuevos naipes al mazo. Además, se deberá registrar, en la clase ***Naipe***, el último naipe creado (una referencia al objeto ***Naipe*** correspondiente) el cual podrá ser consultado utilizando un método correspondiente. Utilizar la clase ***Mazo*** para crear varios naipes de manera que se pueda observar en cualquier momento el último naipe agregado.
Sobrescribir el método *toString()* en las clases ***Mazo*** y ***Naipe*** para que muestre su contenido y su valor respectivamente.

Agregar un método que permita mezclar el mazo (consultar la clase java.util.*Collections* del *API* de Java).