Práctica 2

Ejercicio 1.

Declaración e implementación de Interface

a) ¿Son correctas las siguientes declaraciones?

```
interface ColPrimarios {
      int ROJO=1, VERDE=2, AZUL=4;
}
interface ColArcoIris extends ColPrimarios {
      int AMARILLO=3, NARANJA=5, INDIGO=6, VIOLETA=7;
}
interface ColImpresion extends ColPrimarios {
      int AMARILLO=8, CYAN=16, MAGENTA=32;
}
interface TodosLosColores extends ColImpresion, ColArcoIris {
      int FUCSIA=17, BORDO=ROJO+90;
}
class MisColores implements ColImpresion, ColArcoIris {
      public MisColores() {
            int unColor=AMARILLO;
      }
}
```

Es incorrecta la declaración ya que MisColores implementa las interfaces ColImpresion y ColArcoIris que las dos tienen definida a la variable AMARILLO a la cual hace referencia. Esto no sería un error si no usara en el código como ocurre en TodosLosColores.

b) Analice el código de la interface y las clases que la implementan. Determine si son legales o no. En caso de ser necesario, realice las correcciones que correspondan. ¿Cómo podría modificar el método afinar() para evitar realizar cambios en las clases que implementan InstrumentoMusical ?

```
public interface InstrumentoMusical {

	void hacerSonar();
	String queEs();
	void afinar(){}
}
class abstract InstrumentoDeViento implements InstrumentoMusical
{
	void hacerSonar(){
		System.out.println("Sonar Vientos");
	}
	public String queEs() {
		return "Instrumento de Viento";
	}
}
class InstrumentoDeCuerda implements InstrumentoMusical {
	void hacerSonar(){
		System.out.println("Sonar Cuerdas");
	}
	public String queEs() {
		return "Instrumento de Cuerda";
	}
}
```

En el caso de la clase abstracta es legal no definir comportamiento para el método afinar() (de la interface) ya que se lo delega a las clases concretas. Si existe un error en el código, pero este es parte de la clase InstrumentoDeCuerda ya que al ser una clase concreta debería implementar el comportamiento de la interfaz.

Para evitar que se realicen cambios en las clases que implementan la interfaz InstrumentoMusical se podría establecer al mismo como un método default que contienen una implementación predeterminada.

Ejercicio 2.

Redefina la clase PaintTest del ejercicio 6 de la práctica 1 de manera de imprimir las figuras geométricas ordenadas de acuerdo al valor de su área. Defina la comparación

entre figuras geométricas usando la siguiente regla: una figura A es menor una figura B si el área de A es menor que el área de B. Use para ordenar el arreglo de figuras los métodos de ordenación disponibles en la clase java.util.Arrays

Ejercicio 3.

Se desea implementar un tipo especial de HashSet con la característica de poder consultar la cantidad total de elementos que se agregaron al mismo. Analice y pruebe el siguiente código de manera de corroborar si realiza lo pedido.

```
public class HashSetAgregados<E> extends HashSet<E> {
	private int cantidadAgregados = 0;
	public HashSetAgregados() {
	}
	public HashSetAgregados(int initCap, float loadFactor) {
		super(initCap, loadFactor);
	}
	@Override
	public boolean add(E e) {
		cantidadAgregados++;
		return super.add(e);
	}
	@Override
	public boolean addAll(Collection<? extends E> c) {
		cantidadAgregados += c.size();
		return super.addAll(c);
	}
	public int getCantidadAgregados() {
		return cantidadAgregados;
```

```
    }
}
```

a) Agregue a una instancia de HashSetAgregados los elementos de otra colección (mediante el método addAll). Invoque luego al método getCantidadAgregados. ¿La clase tiene el funcionamiento esperado? ¿Por qué? ¿Tiene relación con la herencia?

La clase no tiene el funcionamiento esperado puesto que al agregar múltiples elementos de una colección, en la función addAll() de HashSetAgregados se incrementa el contador de la cantidad de elementos con la cantidad de la colección que se está agregando y luego se llama al método addAll() de la superclase, que realiza un llamado al método add(), sobreescrito en la clase HashSetAgregados. Este ultimo incrementa el contador nuevamente por cada elemento de la colección y después llama a la función add() de la superclase, donde se realiza efectivamente el agregado de los elementos al HashSet.

Este funcionamiento incorrecto si tiene relación con la herencia, puesto que el problema está estrechamente relacionado con que el método add() esta sobrescrito por una subclase.

b) Diseñe e implemente una alternativa para HashSetAgregados. ¿Qué interface usaría? ¿Qué ventajas proporcionaría esta nueva implementación respecto de la original?

La interface que usaría es la de Set, la cual define la funcionalidad de la clase HashSet. De esta forma se flexibiliza el diseño y de esta forma el ForwardingSet (la clase wrapper) puede usarse para trabajar con cualquier implementación de Set y funcionará junto con cualquier constructor preexistente.

En definitiva, en lugar de usarse herencia se utiliza composición aplicando el patrón Decorator. La clase existente se convierte en un componente de la nueva, en lugar de ser su superclase. Cada método en la nueva clase invoca el método correspondiente de la clase existente y devuelve los resultados.

La clase resultante va a ser muy sólida, sin dependencias en los detalles de implementación de la clase existente. Además agregar nuevos métodos a la clase existente no tendrá impacto en la nueva clase.

c) Se desea implementar otro tipo especial de Set con la característica de poder consultar la cantidad total de elementos que se removieron del mismo. Diseñe e implemente una solución que permita fácilmente definir nuevos tipos de Set con distintas características.

Ejercicio 4.

Redefina las clases del ejercicio 6 de la práctica 1 de manera que las figuras se puedan serializar.

a) ¿Cómo se serializa un objeto? ¿Con qué fin?

La serialización de un objeto consiste en generar una secuencia de bytes lista para su almacenamiento o transmisión (como enviarlo a otra máquina virtual de Java). Después, mediante la deserialización, el estado original del objeto se puede reconstruir.

Para que un objeto sea serializable se implementa la interfaz java.io.Serializable que lo único que hace es marcar el objeto como serializable, sin que se tenga que implementar ningún método.

b) ¿Qué relación tiene con el serialVersionUID? Analice su impacto al modificar la implementación de las clases.

Cada clase serializable tiene un número de identificación único conocido como serial version UIDs. Si no se especifica este número declarando un campo static final long llamado serialVersionUID, el sistema lo genera automáticamente en tiempo de ejecución aplicando una función hash criptográfica (SHA-1) a la estructura de la clase. Este valor se ve afectado por los nombres de la clase, las interfaces que implementa y la mayoría de sus miembros, incluidos los miembros sintéticos generados por el compilador. Si se cambia alguna de estas cosas, por

ejemplo, agregando un método, el serialVersionUID generado cambia. Si no se declara un serialVersionUID, la compatibilidad se romperá, lo que resultará en una InvalidClassException en tiempo de ejecución.

Un costo importante de implementar Serializable es que reduce la flexibilidad para cambiar la implementación de una clase una vez que haya sido lanzada. Otro costo es que aumenta la probabilidad de errores y vulnerabilidades de seguridad. Además, implementar Serializable incrementa la cantidad de test necesarios al lanzar una nueva versión de la clase.