1.- Declaración e implementación de Interfaces.
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;
   }
}
```

Si. Las declaraciones están bien. Hay conflicto en la asignación de unColor porque no le especificamos qué tomar.

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 instrumento musical, afinar no puede tener cuerpo porque es abstracto y public por difolt
- void afinar(); ← para arreglar punto 1
- Para no modificar las clases que implementan la interfaz habría que implementar el default void afinar(){
  print pitogomoso
  }

canon y radar se mueven juntos
sacar strategy y hacer varios

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?

El problema es que addall suma c.size, pero en el hashset se agregan solamente los elementos no repetidos. Cuenta los intentos de agregar, no los elementos realmente agregados.

La clase no tiene el funcionamiento esperado porque al sobrescribir `addAll` se suma directamente el tamaño de la colección sin verificar si los elementos realmente fueron insertados (por ejemplo, si había duplicados). Esto ocurre porque se invoca a `super.addAll(c)` en lugar de reutilizar el `add` sobrescrito en nuestra clase. Sí tiene relación con la herencia: al heredar y redefinir métodos, debemos tener cuidado de mantener la coherencia entre ellos.

no entendí :D
borrando el addcount dentro de addall me sirve.
No esta bueno hacer herencia sobre cosas que no conozco, me llegan a cambiar un metodo y cague. Si hago composición(implemens, interface).

tener un objeto del tipo que necesito, lo llamo y hace el lo que quiera. no dependo de cómo está implementado en la superclase
tengo problemas si una superclase depende de los detalles de la iomplementación de la superclase para funcionar bien.

en clase: El problema es que al delegar en el addAll no sabemos que hace y lo que hace es llamar a this.add (que llama al que definimos nosotros) entonces se duplica y cada elemento se cuenta dos veces.

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

Una solución es NO heredar de hashset sino usar composición y la interfaz set. no extiendo la clase concreta sino que creo una que envuelva el hashset y delegue los métodos del set al objeto.
antes, cuando llamaba al método addall se llama después al add pero no llama al de la clase de arriba sino que llama al de la clase que implemente yo entonces se agregaban dos veces
Esto es en abstractcollection()

```
See Also:       add(Object)

public boolean addAll( @NotNull Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

haciendo esto

```
public class SetWrapper<E> implements Set<E> {  1 usage  1 inheritor  new *
    private final Set<E> set;  14 usages
```

y que después

```
@Override  Josefina Martinez
public boolean addAll(Collection<? extends E> c) {
  cantidadAgregados += c.size();
  return super.addAll(c);
}
```

Lo que hago es que cuando después el abstractcollection llame a add(e) se llame al del wrapper que va a llamar al de arriba y no al método quie está en el hashset que defini yo

Este patrón rompe la dependencia de los detalles internos de `HashSet` y evita el doble conteo que aparecía cuando se heredaba directamente.
chatgpt ()

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.

```java
public class HashSetEliminados<E> extends SetWrapper<E>{  no usages  new *
    private int cantidadEliminados=0;  3 usages
    public HashSetEliminados(Set<E> set){  no usages  new *
        super(set);
    }
    @Override  new *
    public boolean remove(Object o) {
        cantidadEliminados++;
        return super.remove(o);
    }
    @Override  new *
    public boolean removeAll(Collection<?> c) {
        cantidadEliminados+= c.size();
        return super.removeAll(c);
    }

    public int getCantidadEliminados() {  no usages  new *
        return cantidadEliminados;
    }
}
```

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 en Java es el proceso de convertir un objeto en una secuencia de bytes para que pueda ser almacenado en un archivo, transmitido por la red o guardado en una base de datos.
Se serializa haciendo que implemente la interfaz Serializable y se usa ObjectOutputStream para escribir el objeto y ObjectInputStream para leerlo

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

El serialVersionUID es un identificador de versión de la clase serializable. Sirve para garantizar la compatibilidad entre la clase usada para serializar y la usada para deserializar. Si no coincide, se produce una excepción.
Cuando serializas un objeto, Java incluye el serialVersionUID en los datos serializados. Al deserializar, Java compara:

El serialVersionUID del archivo serializado
El serialVersionUID de la clase actual
Si no coinciden, lanza una InvalidClassException.