Primer Certamen

En este certamen usted no podrá hacer preguntas. Si algo no está claro, indíquelo en su respuesta, haga una suposición razonable y resuelva conforme a ella.

Primera parte, sin apuntes (30 minutos; 32 puntos):

1.- Responda brevemente y entregue en hoja con su nombre.

a) ¿La Programación Orientada a Objeto es un tipo de programación Imperativa o Declarativa? Explique.
La Programación Orientada a Objetos (POO) es Imperativa. En la programación Imperativa el programa señala cómo se debe llevar la computación. Éste es el caso de la POO.
(Nota: En la programación declarativa se indica qué se debe hacer y las condiciones en las cuales se hace, pero no el cómo se debe hacer)

b) Defina con sus palabras los siguientes términos: instancia, clase heredada, principio de sustitución.
Instancia es un objeto o ente específico que cumple con la descripción hecha en una clase.
Clase heredada: es una clase definida a partir de otra más general llamada clase base, padre o superclase. Una clase heredada reutiliza el código y extiende la clase base haciéndola más específica.
Principio de sustitución: es la propiedad de los objetos de poder tomar el lugar donde se esperaba uno de categoría más general. Se da entre instancias de clases derivadas donde se espera una de la base o de jerarquía superior. También se da entre instancias de una clases que toman el lugar donde se espera un objeto que cumpla una interfaz.

c) ¿Por qué se dice que la relación es-un entre clases es una condición necesaria pero no suficiente para identificar relación de herencia entre clases? ¿Qué condición adicional se debe cumplir para que identificar herencia?
Es una condición necesaria pues la relación de herencia sólo se da cuando existe la relación es-un entre una instancia de la clase heredada y una instancia de la clase base. No es suficiente pues existen casos de relación es-un pero donde no hay herencia. Un ejemplo de esto es: un cuadrado es un rectángulo de lados iguales. Existe relación es-un pero no herencia.
La condición adicional es cumplir el principio de sustitución. En el ejemplo previo el comportamiento de un rectángulo nos permite ensancharlo hasta que su ancho sea, por ejemplo, el doble de su alto. Si en su lugar usáramos un cuadrado, este comportamiento no es posible.

d) ¿Qué hace posible que un mismo byte-code pueda correr en máquinas de distintas arquitecturas y distintos sistemas operativos?
Esto se logra por la existencia de una Máquina Virtual Java (JVM) para cada par (arquitectura, sistema operativo).

e) Mencione dos usos de la palabra reservada "final".
"final" lo usamos para identificar que un atributo no puede cambiar su valor una vez asignado en su constructor. "final" también lo usamos para indicar que un método no puede ser redefinido en clases derivadas.

f) Mencione dos semejanzas entre clases abstractas e interfaces en Java. Mencione una diferencia entre ambas.
Semejanzas: No podemos crear instancias de clases abstractas ni de interfaces. Las clases abstractas y las interfaces poseen métodos no implementados.
Diferencia: Una clase pude derivar de sólo una clase abstracta, pero puede implementar múltiples interfaces.

g) ¿Qué imprime el programa Test?

```
public class A{
  public A(){};
  public void quienSoy() {
    System.out.println("Soy A");
  }
}
```

```
public class B extends A{
  public B(){};
  public void quienSoy() {
    System.out.println("Soy B");
  }
}
```

```
public class Test{
 public static void main(String args[]){
```

```
        A a = new B();
        a.quienSoy();
        B b = (B) a;  /* ésta fue una omisión, sin ella hay error de compilación*/
        b.quienSoy();
        A aa = (A)a;
        aa.quienSoy();
    }
   }
```

El "cast" en A aa = (A) a; pudo omitirse.
Imprime:
Soy B
Soy B
Soy B

h) ¿Qué se entiende por copia baja y copia profunda? ¿Qué tipo de copia implementa el método clone() de la clase Object?
Copia baja es aquella donde la copia de un objeto se efectúa copiando el valor de sus atributos y cuando un atributo es un objeto sólo se copia su referencia, como consecuencia en copia baja los atributos objetos son compartidos entre el objeto original copiado y su destino.
Copia profunda es aquella donde la copia de un objeto sólo copia los atributos escalares (no objetos) y los objetos inmutables (que no pueden cambiar) y para aquellos objeto mutables se generan copias profundas de cada uno. De este modo se genera una nueva versión del objeto que no comparte objetos mutables con el objeto copiado.
El método clone() implementa copia baja.

Segunda Parte, con apuntes (60 minutos)

2.- (28 puntos) Reescriba el programa adjunto para crear una versión que no haga uso de clases internas ni clases anónimas.

```
public class CreaBotones {
  public static void main(String[] args) {
    CreaBotonesFrame frame = new CreaBotonesFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
class CreaBotonesFrame extends JFrame {
  public CreaBotonesFrame() {
    setSize(250, 150);
    CreaBotonesPanel panel = new CreaBotonesPanel();
    Container contentPane = getContentPane();
    contentPane.add(sp);
  }
}
class CreaBotonesPanel extends JPanel {
  public CreaBotonesPanel() {
    creaBoton();
  }
  private void creaBoton() {
    JButton boton = new JButton("Crea Boton");
    boton.addActionListener( new ActionListener (){
       public void actionPerformed(ActionEvent event) {
         creaBoton();
       };
      });
    add(boton);
    validate();
  }
}
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class CreaBotones {
  public static void main(String[] args) {
    CreaBotonesFrame frame = new CreaBotonesFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
class CreaBotonesFrame extends JFrame {
  public CreaBotonesFrame() {
    setSize(250, 150);
    CreaBotonesPanel panel = new CreaBotonesPanel();
    Container contentPane = getContentPane();
    contentPane.add(panel);
  }
}
class CreaBotonesPanel extends JPanel {
  public CreaBotonesPanel() {
    creaBoton();
  }
  public void creaBoton() {
    JButton boton = new JButton("Crea Boton");
    boton.addActionListener( new CreaBotonListener (this));
    add(boton);
    validate();
  }
}

class CreaBotonListener implements ActionListener {
   private CreaBotonesPanel creaBotones;
   CreaBotonListener (CreaBotonesPanel cbp) {
     creaBotones = cbp;
   }
   public void actionPerformed(ActionEvent event) {
     creaBotones.creaBoton();
   }
}
```

3) (40 puntos) En esta pregunta usted explorará el uso de herencia para la generación de código reutilizable. En la Tarea 1 usted debió crear la clase Vector2D. Luego debió cambiar varios métodos de las clases heredadas de PhysicsElement para reflejar la dinámica de los objetos en dos dimensiones. Si ahora se le pidiera una solución en tres dimensiones, una opción es crear la clase Vector3D y repetir los cambios de código hechos en Tarea 1. Alguien propone una opción más general que requeriría cambiar sólo una vez los códigos de clases que hereden de PhysicsElement. Él propone crear una clase "genérica" VectorND y usar objetos VectorND en las clases heredadas de PhysicsElement.

a) Muestre cómo quedaría la clase FixedHook.java al usar sólo objetos VectorND.

```
import java.util.*;
public class FixedHook extends AttachableElement {
  private static int id=0;
  private VectorND pos_t;     // current position at time t
  private Vector<Spring> springs;  // Vector can grow, arrays cannot.
```

```
  private FixedHook(){   // nobody can create a block without state
    super(id++);
  }
  public FixedHook(VectorND pos){
    super(id++);
    pos_t = pos;
    springs = new Vector<Spring>();
  }
  public void attachSpring (Spring spring) {
    springs.add(spring);
  }
  public void detachSpring (Spring spring) {
    springs.remove(spring);
  }
  public VectorND getPosition() {
    return pos_t;
  }
   public String getDescription() { // para generalizar la salida, posición debe
    return "FixedHook_" + getId()+ " "+pos.getDescription(); // indicar su descripción.
  }
  public String getState() {
    return getPosition()+""; // posición debe convertirse a string.
  }
}
```

b) Muestre un código posible para VectorND.

```
public abstract class VectorND {
  public abstract VectorND plus(VectorND v);
  public abstract VectorND times(double scalar);
  public abstract VectorND minus(VectorND v);
  public abstract double module();
  public VectorND unitary() {
    return times(1/module());
  }
  public abstract String getDescription();
}
```

c) Muestre un código posible para la clase Vector1D que se usaría para el caso en una dimensión.

```
public class Vector1D extends VectorND implements Cloneable {
  private double x;  // we will use cartesian coordinates
  public Vector1D () {
    x = 0.0;
  }
  public Vector1D (double x) {
    this.x = x;
  }
  public Vector1D plus(VectorND v) { /* debe mantenerse prototipo para redefinir método */
    try {
      if (v==null) return (Vector1D) clone();
    } catch (CloneNotSupportedException e){
      return null;
    }
    return new Vector1D(x+((Vector1D)v).x);
  }
  public Vector1D times(double scalar) {
    return new Vector1D(x*scalar);
  }
  public Vector1D minus(VectorND v) {
    try {
      if (v==null) return (Vector1D) clone();
    } catch (CloneNotSupportedException e){
      return null;
    }
    return new Vector1D(x-((Vector1D)v).x);
  }
  public double module() {
    return Math.abs(x);
  }
  public String getDescription() {
    return " x ";
  }
  public String toString() {
```

```
    return x+" ";
  }
}
```

d) Comente algunas dificultades, si las observa, de la estrategia propuesta.
Una dificultas es la creación de vectores en el origen. La clase VectorND no permite crear instancias propias por ser abstracta, pero en las implementaciones es conveniente contar con vectores nulos; por ejemplo, al inicializar las fuerzas se debe partir con fuerza nula y luego ir agregando fuerzas. Si no contamos con vector nulo, podríamos manejar un objeto null cuando el vector sea nulo, pero esta estrategia obliga a verificar la existencia del vector antes de hacer operaciones como suma.
Una solución alternativa es generar un vector nulo a partir de la resta de un vector consigo mismo, pero esta "solución" sólo sirve cuando se tiene una referencia a vector no null.

```
public class FixedHook extends AttachableElement  {
   private static int id=0;
   private double pos_t;     // current position at time t
   private Vector<Spring> springs;  // Vector can grow, arrays cannot.

   private FixedHook(){   // nobody can create a block without state
     super(id++);
   }
   public FixedHook(float hight){
      super(id++);
      pos_t = hight;
      springs = new Vector<Spring>();
   }
   public void attachSpring (Spring spring) {
     springs.add(spring);
   }
   public void detachSpring (Spring spring) {
     springs.remove(spring);
   }
   public double getPosition() {
      return pos_t;
   }
   public String getDescription() {
      return "FixedHook_" + getId()+": y";
   }
   public String getState() {
      return getPosition()+"";
   }
}
```