COMPLEMENTOS DE PROGRAMACIÓN

EXAMEN JUNIO 2013

1. **Explica la ventaja de controlar los posibles errores de ejecución de un programa mediante excepciones frente a la forma clásica que se usa en lenguajes como C.**

Al poder controlar los posibles errores de ejecución de un programa mediante excepciones podemos separar la detección de errores del manejo de ellos. Es decir, la ventaja de usar excepciones es que un método puede enviar una excepción a su método llamante, permitiendo a éste manejar la excepción.

2. **Indica qué restricción existe a la hora de ordenar múltiples cláusulas catch de un bloque try-catch, y explica por qué.**

Como cada bloque catch se examine en el orden en el que esté escrito desde el primero al último, no puede aparecer antes un catch con una excepción de una superclase antes que un catch con una excepción de una de sus subclases. Este hecho provocaría un error.

Este error se debe a que si capturamos con el bloque catch una excepción de una superclase, significa que estamos capturando todas las excepciones que esa superclase pueda contener. Por tanto si después ponemos el catch con la excepción de una subclase, saltará el error advirtiéndonos que ya se ha capturado esa excepción en el bloque catch anterior.

3. **Suponiendo la clases y los programas dados, explicar qué ocurre y porqué.**
    a) **Programa 1.**

    El resultado del programa es:

    "ClaseB.metodo1 ( )"

    Cuando se llama al método 1 del objeto A, como este objeto está apuntando a un objeto B y en la clase B está sobrescrito el método "metodo1 ()", se ejecuta el método 1 de la clase B.

    b) **Programa 2.**

    El programa no imprime ningún resultado ya que no puede llamarse al método 2 del objeto A. Esto es debido a que la clase A no implementa el método 2, es decir, método 2 es un método perteneciente exclusivamente a la clase B. Solo se podrá ejecutar el método en este caso:

    ClaseB objetoB = new ClaseB ();

    objetoB.metodo2();

4. **Explica qué pasa si en una subclase se redefine un método estático que estaba definido en una superclase. O sea, el método se implementa en ambas clases con el mismo nombre, mismos parámetros, mismo tipo devuelto, pero con cuerpo diferente.**

Un método static no puede sobrescribirse pero si se redifine el método de la súper clase, quedará oculto. Es decir, si quisiéramos utilizar el método static de la súper clase deberíamos llamarlo desde el nombre de dicha superclase:

NombreSuperclase.metodoEstatico (---)

5. **Dado el código de Java, escribe una clase que permita guardar una colección de objetos cuya clase pueda ser cualquier subclase de Figura. Debe contener:**
    - **Un método para añadir un nuevo objeto (de cualquier subclase de Figura) a la colección.**
    - **Un método para recuperar el elemento de la posición i.**
    - **Un método para imprimir en la salida estándar el área de todos los elementos de la colección. Para ello, añade también a las siguientes clases el código necesario para poder obtener el área.**

La clase nueva creada quedaría del siguiente modo:

```
import java.util.ArrayList;

public class Coleccion {

   ArrayList<Figura> Figuras;

   public Coleccion(int dimension){

      Figuras = new ArrayList<>(dimension);

   }


   void anadirFigura(Figura figura){

      Figuras.add(figura);

   }


   Figura recuperarFigura(int posicion){

      Figura figura = Figuras.get(posicion);

      return figura;

   }
```

```
    void imprimirAreaFiguras(){
        double AreaRectangulos = 0;
        double AreaTriangulos = 0;
        double AreaTotal;

        for(int i = 0; i < Figuras.size(); i++){
            if(Figuras.get(i) instanceof Rectangulo){
                System.out.println("Area del Rectangulo: " + Figuras.get(i).getArea());
            }
            else{
                System.out.println("Area del Triangulo: " + Figuras.get(i).getArea());
            }
        }
    }
}
```

//

Las clases del código del examen quedarían tal que:

```
abstract class Figura {
    double dim1, dim2;
    Figura(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    public abstract double getArea();
}
class Rectangulo extends Figura {
    double ancho;
    double alto;
    Rectangulo(double a, double b) {
```

```
        super(a, b);
        this.ancho = a;
        this.alto = b;
    }
    public double getArea(){
        return ancho * alto;
    }
}
class Triangulo extends Figura {
    double ancho;
    double alto;
    Triangulo(double a, double b) {
        super(a, b);
        this.ancho = a;
        this.alto = b;
    }
    public double getArea(){
        return (ancho * alto) / 2;
    }
}
```

//

6. **Construir un programa en el que se implemente el problema del producto/consumidor, ene l que una hebra Productor y una hebra Consumidor se ejecutan concurrentemente con la hebra principal. La hebra Productor escribe números enteros en un buffer y la hebra Consumidor lee datos del mismo buffer. Debe asegurarse la correcta sincronización del productor y consumidor en el acceso al buffer compartido. Además, debemos asegurar que cada dato es consumido una sola vez por el consumidor y que el consumidor no consume datos si el buffer está vacío. También debemos asegurar que el productor no produce datos cuando el buffer está lleno. El buffer almacenará los datos enteros en un array. El tamaño del array se determinará con un parámetro del constructor de la clase a la que pertenece el buffer. Como mecanismo de sincronización entre**

productor y consumidor podrá usarse únicamente los métodos wait, notify y notifyAll.

Para resolver este problema se construirán las siguientes clases:

a) **Clase Buffer: Debe implementarse para que los datos se guarden en un array de enteros (cuya capacidad se definirá mediante un parámetro en el constructor de la clase). Añada además los datos miembro necesarios. No se permite el uso de una cola bloqueante. Además debe contener un método get para obtener el siguiente entero del buffer y un método set para añadir un entero al buffer.**
b) **Clase Productor: Representa la tarea ejecutada por la hebra productora. La función de esta tarea es insertar los números de 0 a 99 en el buffer compartido.**
c) **Clase Consumidor: Representa la tarea ejecutada por la hebra consumidora. La función de esta tarea es obtener la suma de 100 números obtenidos del buffer compartido.**
d) **Clase ProgramaPrincipal: Contiene la función main. Debe crear los objetos necesarios para que funcione el programa y se debe encargar de crear y lanzar las hebras productor y consumidor. El Buffer se creará con una capacidad especificada en el primer parámetro de main.**

Aunque no nos lo pide el programa, para la case Buffer debemos hacer una interfaz donde tengamos los métodos que va a usar el buffer compartido. Por tanto la interfaz sería:

```
public interface Buffer {
   public void setInt(int entero) throws InterruptedException;
   public int getInt() throws InterruptedException;
}
```

//

La implementación del Buffer sería:

```
public class BufferImplementacion implements Buffer {
   int[] arrayEnteros;
   private int arrayPosicion = 0;
   public BufferImplementacion(int dimension) {
      arrayEnteros = new int[dimension];
   }
```

```
    @Override
    public synchronized void setInt(int entero) throws InterruptedException {
        while (arrayPosicion >= arrayEnteros.length) {
            System.out.println("Buffer lleno");
            wait();
        }
        arrayEnteros[arrayPosicion] = entero;
        System.out.println("Productor escribe: " + entero);
        arrayPosicion++;
        notifyAll();
    }
    @Override
    public synchronized int getInt() throws InterruptedException {
        int[] auxiliar = new int[arrayEnteros.length - 1];
        int entero;
        while (arrayPosicion == 0) {
            System.out.println("Buffer vacio");
            wait();
        }
        entero = arrayEnteros[0];
        System.arraycopy(arrayEnteros, 1, auxiliar, 0, (arrayEnteros.length - 1));
        arrayPosicion--;
        System.arraycopy(auxiliar, 0, arrayEnteros, 0, auxiliar.length);
        notifyAll();
        return entero;
    }
}
////////////////////////////////////////////////////////////////////////////////////////
```

La clase Productor sería:

```
public class Productor implements Runnable {
   private final Buffer buffer;
   public Productor(Buffer buf) {
      this.buffer = buf;
   }
   public void run() {
      try {
         for (int i = 0; i < 100; i++) {
            buffer.setInt(i);
         }
      } catch (InterruptedException e) {
         System.out.print("Error capturado");
      }
   }
}
```

//

La clase Consumidor sería:

```
public class Consumidor implements Runnable{
   private Buffer buffer;
   private int suma = 0;
   public Consumidor(Buffer buf){
      this.buffer = buf;
   }
   public void run(){
      try{
         for(int i = 0; i < 100; i++){
            suma += buffer.getInt();
            System.out.println("SUMA: " + suma);
```

```
        }
      } catch (InterruptedException e){
        System.out.println("Interrupcionn capturada");
      }
    }
}
```

//

Y por último el programa principal sería:

```
public class ProgramaPrincipal {
    public static void main(String[] args) {
      int dimension = 5;
      //Creamos el pool de hebras con las hebras que sean necesarias
      ExecutorService programa = Executors.newCachedThreadPool();
      //Creamos el buffer compartido entre productor y consumidor
      Buffer bufferCompartido = new BufferImplementacion(dimension);
      //Creamos las hebras Consumidor y Productor
      programa.execute(new Productor(bufferCompartido));
      programa.execute(new Consumidor(bufferCompartido));
      //Finalizamos el pool de hebras
      programa.shutdown();
    }
}
```

//