COMPLEMENTOS DE PROGRAMACIÓN

EXAMEN SEPTIEMBRE 2013

1. **Indica bajo qué condición proporciona Java automáticamente un constructor por defecto.**

Java proporciona un constructor por defecto sin argumentos que inicializa los valores de los datos miembro a 0 automáticamente cuando no se definen constructores de forma explícita.

2. **Muestra la salida del programa dado:**

La salida del programa es:

i en clase A es 60

i en clase B es 60

Cuando se llama a "new B()" se llama al constructor de B que como es una clase heredada de A se pasa a llamar al constructor de la clase A y se ejecuta el método setI(i) de la clase B ya que está sobrescrito en dicha clase. Tras esto se ejecuta el system.out de la clase A y cuando termina se vuelve al constructor de la clase B y se ejecuta el system.out de dicha clase.

3. **En un bloque try-catch, indica qué pasa si ocurre una excepción cuando se está ejecutando una sentencia de la cláusula catch.**

Si dicha excepción está en otro bloque try-catch se capturará dicha excepción mientras que si no existe ese bloque nuevo try catch, el programa devolvería un error en tiempo de ejecución y volvería al método llamante.

4. **¿Para qué sirve la sentencia throw referenciaaExcepcion?**

Sirve para lanzar la excepción llamada "referenciaaExcepcion" si ocurriera este tipo de excepción durante la ejecución del problema.

5. **Un número racional tiene un numerador y un denominador (enteros long) en la forma a/b, donde a es el numerador y b el denominador. Un racional no puede tener 0 como denominador. En la clase que debemos construir, representaremos el numerador y el denominador usando números enteros. Escribe la clase Racional siguiendo las siguientes indicaciones:**
    a) **Declara la clase Racional como una subclase de la clase Number e incluye los datos miembro necesarios. La clase debe implementar también el interfaz genérico Comparable para poder comparar dos números racionales.**
    b) **Añade el constructor Racional (): crea un racional con 0 como numerador y 1 como denominador.**
    c) **Incluye un método estático para calcular el máximo común divisor de dos números enteros de tipo long siguiendo el algoritmo de Euclides**

d) **Añade el constructor Racional (long numerador, long denominador): crea un racional con el numerador y denominador especificados. Impleméntalo de forma que numerador y denominador queden reducidos a sus términos más pequeños, haciendo uso del máximo común divisor.**
e) **Sobrescribe el resto de métodos de Number (los métodos abstractos). Estos métodos convierten el Racional al tipo correspondiente mediante la división del numerador y el denominador.**
f) **Sobrescribe el siguiente método de la clase Object para comprobar si este racional es igual al especificado. "public boolean equals (Object racional2)".**
g) **Sobrescribe el siguiente método de la clase Object para devolver un String a partir de este racional. El string tendrá la forma a/b si b es distinto de 1, y la forma a si b es igual a 1. "public String toString()".**
h) **Supongamos que disponemos de un método que lee un número (de alguna de las subclases de Number) de la entrada estándar y lo devuelve: "public static Number leeNumero()" Este método podrá devolver objetos de cualquier subclase de Number. Construir un programa que lea 100 números usando el método anterior y los almacene en un array.**

El programa quedaría así sin el último apartado:

```
//  a)
public class Racional extends Number implements Comparable<Number>{
   private final long a;
   private final long b;
//  b)
   Racional(){
      this.a = 0;
      this.b = 1;
   }
//  d)
   public Racional(long numerador, long denominador) {
      this.a = numerador;
      this.b = denominador;
   }
//  c)
```

```
    public static long MCD(long a, long b){
        if(b == 0){
            return a;
        }
        else if(a%b != 0){
            long c = a%b;
            MCD(a, c);
        }
        return a;
    }
//    e)
    @Override
    public int intValue() {
        return (int) (this.a / this.b);
    }
    @Override
    public  long longValue(){
        return this.a / this.b;
    }
    @Override
    public float floatValue(){
        return (float) (this.a / this.b);
    }
    @Override
    public double doubleValue(){
        return (double) (this.a / this.b);
    }
     @Override
    public int compareTo(Number o) {
        int resultado = 0;
```

```
        long racional = longValue();
        if(racional < o.longValue()){
            resultado = -1;
        }else if(racional > o.longValue()){
            resultado = 1;
        }
        return resultado;
    }
//    f)
    /************MADEBYJUSTICIA*************/
    @Override
    public boolean equals(Object racional2) {
        if(racional2 instanceof Racional){
            return ((this.a == ((Racional)racional2).a) && (this.b == ((Racional)racional2).b));
        }
        else{
            return false;
        }
    }
//    g)
    @Override
    public String toString(){
        if(this.b != 1){
            long racional = this.a/this.b;
            return "" + racional;
        }
        else{
            return "" + this.a;
        }
    }
```

```
////////////////////////////////////////////////////////////////////////////////
```

Apartado h (se supone que el método leeNumero() está hecho):

```
public class Apartadoh {
   private static final Number[] array = new Number[100];
   public static void main (String[] args){
      for(int i = 0; i < array.length; i++){
         array[i] = leeNumero();
      }
   }
}
////////////////////////////////////////////////////////////////////////////////
```

6. **Construir un programa en el que se implemente el problema del producto/consumidor, en el 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 de 5.

Clase Buffer:

```
public interface Buffer{
   public int get() throws InterruptedException;
   public void set(int valor) throws InterruptedException;
}
```

```
////////////////////////////////////////////////////////////////////////////////
```

Clase BufferImpl:

```
public class BufferImpl implements Buffer {
   private final int[] buffer;
   private int posBuffer = 0;
   BufferImpl(int dimension) {
      buffer = new int[dimension];
   }
   @Override
   public synchronized int get() throws InterruptedException {
      int entero = 0;
      int[] auxiliar = new int[buffer.length];
      try {
         while(posBuffer == 0){
            System.out.println("Buffer vacio");
            wait();
         }
//          Solo puede leer un dato el consumidor, el primero del buffer
         entero = buffer[0];
         System.arraycopy(buffer, 1, auxiliar, 0, (buffer.length-1));
         System.arraycopy(auxiliar, 0, buffer, 0, auxiliar.length);
         System.out.println("Consumidor lee: " + buffer[0]);
```

```
            posBuffer--;
            notifyAll();
        } catch (InterruptedException ex) {
            System.out.println("Excepcion en get()" + ex.toString());
        }
        return entero;
    }
    @Override
    public synchronized void set(int valor) throws InterruptedException {
        try {
            while (posBuffer >= buffer.length) {
                System.out.println("Buffer lleno");
                wait();
            }
            buffer[posBuffer] = valor;
            posBuffer++;
            notifyAll();
            System.out.println("Productor escribe: " + valor);
        } catch (InterruptedException ex) {
            System.out.println("Excepcion en set()" + ex.toString());
        }
    }
}
////////////////////////////////////////////////////////////////////////////////////////
```

Clase Productor:

```
public class Productor implements Runnable {
    private Buffer buffer;
    Productor(Buffer buf){
        this.buffer = buf;
```

```
    }
    @Override
    public void run() {
        try {
            for (int i = 0; i < 100; i++) {
                buffer.set(i);
            }
        } catch(Exception e){
            System.out.println("Excepcion en Productor " + e.toString());
        }
    }
}
```

//

Clase Consumidor:

```
public class Consumidor implements Runnable{
    private final Buffer buffer;
    Consumidor(Buffer buf){
        this.buffer = buf;
    }
    @Override
    public void run(){
        int suma = 0;
        try{
            for(int i = 0; i < 100; i++){
                suma += buffer.get();
                System.out.println(suma);
            }
        } catch(Exception e){
            System.out.println("Excepcion en Consumidor " + e.toString());
        }
```

```
    }
}
```

//

Clase ProgramaPrincipal:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ProgramaPrincipal {
    public static void main(String[] args){
        Buffer buffer = new BufferImpl(5);
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.execute(new Productor(buffer));
        executor.execute(new Consumidor(buffer));
        executor.shutdown();
    }
}
```

//

7. En la clase MetodosGenericos, implementa un método genérico para imprimir en la salida estándar cada uno de los objetos almacenados en un array de un tipo genérico.

El método sería:

```
public class MetodosGenericos {
    public void imprimeObjetos(ArrayList<Integer> lista){
        for(int i = 0; i < lista.size(); i++){
            System.out.println(lista.get(i));
        }
    }
}
```

//

Indica cómo se usaría este método para imprimir todos los números almacenados en el array construido en el ejercicio 5h.

```
public class Apartadoh {
   private static final ArrayList<Number> array = new ArrayList<>();
   public static void main (String[] args){
      for(int i = 0; i < 100; i++){
         array.set(i, leeNumero());
      }
      imprimeObjetos(array);
   }
}
////////////////////////////////////////////////////////////////////////////////////////////////
```