EXAMEN CON SOLUCIONES

Duración: 90 minutos
Puntuación máxima: 7 puntos
Fecha: 13 de marzo de 2018
Instrucciones para el examen:

- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.
- Rellena tus datos personales antes de comenzar a realizar el examen.
- Escribe tus respuestas dentro de los recuadros al efecto. Puedes usar hojas adicionales para escribir en sucio que no tendrás que entregar.

APELLIDOS: _____

NOMBRE: _____ NIA: _____ GRUPO: _____

Problema 1. Orientación a Objetos (5 / 7 puntos)

Nos piden simular el funcionamiento de un banco. En el sistema de información del banco existen los siguientes conceptos con la siguiente información asociada:

- Clientes, con nombre y apellidos, y DNI; y lista de cuentas de las que es titular.
- Cuentas corrientes, con número de cuenta, cliente titular asociado, saldo inicial y lista de movimientos.
- Movimientos, con cuenta origen, cuenta destino e importe.
- ClientesVIP: es un tipo de cliente muy importante para el banco, para el cual, se le permite tener descubiertos en sus cuentas, sin límite.

Para la implementación del sistema, es suficiente tratar los importes y saldos como tipo primitivo `double`; y los números de cuenta como de tipo `int`. Todos los atributos deberán ser privados, pero accesibles mediante sus correspondientes *getters* para su consulta. A continuación se muestra el código de todas las clases, que se deben leer y comprender previamente.

```java
public class Cliente {
    private String nombre;
    private String apellidos;
    private String dni;
    private List<Cuenta> cuentas;
    public Cliente(String nombre, String apellidos, String dni) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.dni = dni;
        this.cuentas = new ArrayList<Cuenta>();
    }
    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos; }
    public String getDni() { return dni; }
    public List<Cuenta> getCuentas() { return cuentas; }
    public Cuenta abrirCuenta(double saldoInicial) {
        // IMPLEMENTAR EN APARTADO 1.2
    }
    public void imprimirSituacion() {
        System.out.println(this.nombre + " " + this.apellidos + " es un cliente con las siguientes cuentas:");
        for (int i = 0; i < this.cuentas.size(); i++) {
            this.cuentas.get(i).imprimirSituacion();
        }
    }
}
```

```java
public class ClienteVIP extends Cliente {
    public ClienteVIP(String nombre, String apellidos, String dni) { super(nombre, apellidos, dni); }
    @Override
    public void imprimirSituacion() {
        // IMPLEMENTAR EN APARTADO 1.4
```

```
    }
}
```

```
public class SaldoInsuficienteException extends Exception { }
```

```
public class Movimiento { /* IMPLEMENTAR EN APARTADO 1.1 */ }
```

```
public class Cuenta {
    private static int proximoNumeroCuenta = 1;
    private int numeroCuenta;
    private Cliente titular;
    private double saldoInicial;
    private List<Movimiento> movimientos;
    public Cuenta(Cliente titular, double saldoInicial) {
        this.numeroCuenta = proximoNumeroCuenta++;
        this.titular = titular;
        this.saldoInicial = saldoInicial;
        this.movimientos = new ArrayList<Movimiento>();
    }
    public int getNumeroCuenta() { return numeroCuenta; }
    public double getSaldoInicial() { return saldoInicial; }
    public List<Movimiento> getMovimientos() { return movimientos; }
    public Cliente getTitular() { return titular; }
    public double getSaldoActual() {
        // IMPLEMENTAR EN APARTADO 1.5
    }
    public void imprimirSituacion() {
        System.out.println(" Cuenta #" + this.numeroCuenta + ". Saldo actual: " + String.format("%,.02f EUR",
        ↪  this.getSaldoActual()));
    }
    public void transferir(Cuenta destino, double importe) throws SaldoInsuficienteException {
        if (this == destino) {
            throw new IllegalArgumentException("Las cuentas origen y destino no pueden ser iguales");
        }
        if (importe <= 0.0) {
            throw new IllegalArgumentException("El importe debe ser positivo");
        }
        if (importe <= this.getSaldoActual() || this.titular instanceof ClienteVIP) {
            Movimiento m = new Movimiento(this, destino, importe);
            this.movimientos.add(m);
            destino.movimientos.add(m);
        } else {
            throw new SaldoInsuficienteException();
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Cliente juan = new Cliente("Juan", "Hernandez", "24723874X");
        ClienteVIP florentino = new ClienteVIP("Florentino", "Perez", "74824752Z");
        Cuenta juan_cuenta1 = juan.abrirCuenta(500.00);
        Cuenta florentino_cuenta1 = florentino.abrirCuenta(3000.00);
        try {
            florentino_cuenta1.transferir(juan_cuenta1, 3500.00);
        } catch (SaldoInsuficienteException ex) {
            System.out.println("!!! Saldo insuficiente para la operación de Florentino.");
        }
        System.out.println(" ***** Situación final *****");
        juan.imprimirSituacion();
        florentino.imprimirSituacion();
    }
}
```

```
 ***** Situación final *****
Juan Hernandez es un cliente con las siguientes cuentas:
```

```
 Cuenta #1. Saldo actual: 4,000.00 EUR
Florentino Perez es un cliente VIP con las siguientes cuentas:
 Cuenta #2. Saldo actual: -500.00 EUR
```

Nota: no se preocupe de escribir los imports. Si lo necesita, la clase `ArrayList<E>` implementa la interfaz `List<E>` y tiene los siguientes métodos, entre otros:

`boolean add(E e)`	`void add(int index, E element)`	`void clear()`	`E get(int index)`
`int indexOf(Object o)`	`boolean isEmpty()`	`boolean remove(Object o)`	`int size()`

Resuelva los siguientes problemas:

1. 1 punto Implemente la clase `Movimiento`, con dos atributos de tipo `Cuenta` (llamados `origen` y `destino`) y un atributo de tipo `double` llamado `importe`, todos privados. Añada un constructor que dados los tres argumentos (en ese orden) inicialice los tres atributos. Añada los correspondientes *getters* para permitir la lectura de los valores, pero no los *setters*.

Solución:

```java
public class Movimiento {
    private Cuenta origen;
    private Cuenta destino;
    private double importe;
    public Movimiento(Cuenta origen, Cuenta destino, double importe) {
        this.origen = origen;
        this.destino = destino;
        this.importe = importe;
    }
    public Cuenta getOrigen() { return origen; }
    public Cuenta getDestino() { return destino; }
    public double getImporte() { return importe; }
}
```

Criterios de corrección

- Declaración correcta de los tres atributos (0,25)
- Declaración correcta del constructor (0,25)
- Implementación correcta del constructor (0,25)
- Declaración e implementación correcta de los tres *getters* (0,25)
- Se considerarán incorrectos los cambios en los nombres de los atributos, nombre de los métodos y visibilidades. El enunciado lo indica muy explícitamente.

2. 1 punto Implemente el siguiente método de la clase `Cliente`: `public Cuenta abrirCuenta(double saldoInicial)` que deberá crear una nueva cuenta cuyo titular sea el cliente que recibe el mensaje, y el saldo inicial el indicado por parámetro. Se deberá añadir dicha cuenta a la lista de cuentas del titular, y además devolver la cuenta recién creada.

Solución:

```java
    public Cuenta abrirCuenta(double saldoInicial) {
        Cuenta c = new Cuenta(this, saldoInicial);
        this.cuentas.add(c);
        return c;
    }
```

Criterios de corrección

- Creación de nuevo objeto `Cuenta` con los valores correctos (0,25)

- Asignación a variable local (0,25)
- Adición a la lista de cuentas del `Cliente` (0,25). No importa si se añade al final o si se inserta al principio, pero sin perder alguna posible cuenta que estuviera previamente en la lista.
- Devolución de la nueva cuenta (0,25)

3. 1 punto Explique cómo se consigue que no existan dos números de cuenta iguales para cuentas diferentes en todo el sistema a lo largo de la vida del programa.

Solución: En la clase `Cuenta`, existe el atributo `private static int proximoNumeroCuenta`, que se inicializa al valor 1. En el constructor de cuentas, éste valor se asigna al número de cuenta de la cuenta que se está construyendo, y se postincrementa para la siguiente cuenta. Al ser un atributo estático de la clase y privado, se preservan dichos incrementos entre una y otra invocaciones del constructor, y además se evita que el valor del contador de cuentas pueda ser modificado desde código externo a la clase.

Criterios de corrección

- Referirse a la variable `proximoNumeroCuenta` que sirve de contador (0,25)
- Referirse a su carácter de `static` para preservar el valor (0,25)
- Indicar que se protege la modificación externa gracias a ser `private` (0,25)
- Indicar que el contador se utiliza y se autoincrementa en el constructor de `Cuenta` (0,25)

4. 1 punto Implemente el siguiente método de la clase `ClienteVIP`: `public void imprimirSituacion()` sabiendo que debe imprimir lo mismo que en el caso de `Cliente` pero indicando que es un «cliente VIP» en lugar de «cliente» normal (véase el ejemplo de la salida de la ejecución).

Solución:

```
1     public void imprimirSituacion() {
2         System.out.println(this.getNombre() + " " + this.getApellidos() + " es un cliente VIP con las
          ↪  siguientes cuentas:");
3         for (int i = 0; i < this.getCuentas().size(); i++) {
4             this.getCuentas().get(i).imprimirSituacion();
5         }
6     }
```

Criterios de corrección

- Resolverlo correctamente de manera similar (1,00). Se proporciona la implementación en la superclase precisamente para que se copie y únicamente se reemplace la palabra `"cliente"` por `"cliente VIP"`.
- Llamar a `super.imprimirSituacion()` (-0,25). En este caso, no es utilizable tal cual.
- Utilizar alguno/s de los atributos privados de la superclase sin usar los *getters* (-0,25). Daría un error de acceso.
- Construir y devolver el `String` (siendo éste un método `void`) en lugar de imprimir en salida estándar (-0,25). Se pide que se imprima, no que se devuelva. Además, el método no puede devolver nada porque está declarado `void`.

5. 1 punto Implemente el siguiente método de la clase `Cuenta`: `public double getSaldoActual()` que calcule el saldo actual de esta cuenta. Para ello deberá tener en cuenta el saldo inicial de apertura de la cuenta, y cada uno de los movimientos que se han realizado desde y hacia ésta, que se encuentran en la lista de movimientos de dicha cuenta. Los movimientos asociados cuyo origen sea la cuenta actual se tomarán en sentido negativo (salidas de dinero), y los movimientos cuyo destino sea la cuenta actual se entenderán en sentido positivo (entradas de dinero), y finalmente se devolverá el saldo actual calculado.

Solución:

```
1     public double getSaldoActual() {
2         double resultado = this.saldoInicial;
3         for (int i = 0; i < this.movimientos.size(); i++) {
4             Movimiento m = this.movimientos.get(i);
5             if (m.getDestino() == this) {
6                 resultado = resultado + m.getImporte();
7             } else if (m.getOrigen() == this) {
8                 resultado = resultado - m.getImporte();
9             }
10        }
11        return resultado;
12    }
```

Criterios de corrección

- Usar un acumulador (0,20)
- Tener en cuenta el saldo inicial de la cuenta (0,20)
- Iterar correctamente (en bucle `for(:)` o `for(;;)`) a través de todos los movimientos de la cuenta (0,20)
- Sumar los movimientos destino y restar los movimientos origen (0,20)
- Devolver el acumulador (0,20)
- Usar algún/os atributo/s privado/s sin usar su *getter* (-0,20)

Problema 2. Pruebas de Programa (2 / 7 puntos)

Supongamos la clase `RainyDays`, que indica los días lluviosos a lo largo del año. Estos días se representan con un array de tipo `boolean`, donde si el elemento que ocupa la posición `i` es `true` se interpreta que el día `i` fue un día lluvioso, siendo el primer día del año el día número 0. La clase, está formada por un constructor y por diversos métodos que posibilitan lo siguiente:

- `public void recordRainyDay(int dayOfYear, boolean rained)`: registra el día del año que llovió.
- `public boolean askIfTheDayRained(int dayOfYear)`: devuelve si un día determinado del año llovió.
- `public int countRainyDays()`: devuelve el número de días del año que ha llovido.
- `public int getFirstDayWithRain()`: devuelve el primer día del año que llovió.
- `private static void validateDay(int dayOfYear)`: método para validar que el número de día que introduce un usuario es correcto.

Dado el código de la clase:

```
1  public class RainyDays {
2      private boolean[] days = new boolean[365];
3      private static void validateDay(int dayOfYear) {
4          if (dayOfYear < 0 || dayOfYear >= 365)
5              throw new IllegalArgumentException("dayOfYear fuera del rango [0 ; 365)");
6      }
7      public void recordRainyDay(int dayOfYear, boolean rained) {
8          validateDay(dayOfYear);
9          days[dayOfYear] = rained;
10     }
11     public boolean askIfTheDayRained(int dayOfYear) {
12         validateDay(dayOfYear);
13         return days[dayOfYear];
14     }
15     public int countRainyDays() {
16         int rainyDays = 0;
17         for (int i = 0; i < days.length; i++)
18             if (days[i])
19                 rainyDays++;
20         return rainyDays;
21     }
22     public int getFirstDayWithRain() {
```

```
23          for (int i = 0; i < days.length; i++)
24              if (days[i])
25                  return i;
26          return -1;
27      }
28  }
```

Se pide diseñar un caso de pruebas de caja blanca con métodos que comprueben los 4 métodos:

1. [½ punto] `testIfRegisterAllDaysWithoutRain()`: Método que testea si se inicializa correctamente la clase, empezando con cero días lluviosos.
2. [½ punto] `testRecordRainyDay()`: Método que testea si registra correctamente los días lluviosos.
3. [½ punto] `testAskIfTheDayRained()`: Método que testea si devuelve correctamente si un día llovió.
4. [½ punto] `testGetFirstDayWithRain()`: Método que testea si devuelve correctamente el primer día de lluvia.

Solución:

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class RainyDaysTest {

    RainyDays days = new RainyDays();

    @Test
    public void testIfRegisterAllDaysWithoutRain() {
        assertEquals(0, days.countRainyDays());
    }

    @Test
    public void testRecordRainyDay() {
        days.recordRainyDay(5, true);
        assertTrue(days.askIfTheDayRained(5));
    }

    @Test
    public void testAskIfTheDayRained() {
        days.recordRainyDay(10, true);
        assertTrue(days.askIfTheDayRained(10));
    }

    @Test
    public void testGetFirstDayWithRain() {
        assertEquals(-1, days.getFirstDayWithRain());
        days.recordRainyDay(15, true);
        assertEquals(15, days.getFirstDayWithRain());
    }
}
```

Criterios de corrección Para cada uno de los cuatro apartados:

- Si no anota el método con `@Test` (-0,10)
- Si declara correctamente el método (0,10)
- Si utiliza correctamente un objeto de prueba (en el propio método o como atributo de la clase) (0,10)
- Si pone a prueba el método comprobando su valor devuelto esperado con algunos de los métodos de la clase `org.junit.Assert` (0,30)
- No tener en cuenta si faltan los *imports*.