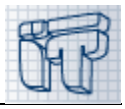


NOMBRE:

APELLIDOS:

NIA:

GRUPO:

Primer examen parcial

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 80 minutos
Puntuación máxima: 7 puntos
Fecha: 9 marzo 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.

Problema 1 (6 puntos)

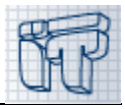
¡Enhorabuena! Has sido elegido para formar parte de un equipo que desarrollará una aplicación para gestionar el Hospital Universitario Carlos III. En concreto, te han encargado el diseño y la implementación de la estructura de clases Java que servirá posteriormente para poder gestionar una base de datos del personal del hospital: doctores, enfermeros, estudiantes en prácticas, etc.

Apartado 1 (2 puntos)

Debes implementar la clase más general: `Persona`. A continuación detallamos qué debes incluir en esa clase:

- **Variables y atributos**: Debes declarar las variables necesarias para guardar `nombre` y año de `nacimiento`. Además, es necesario declarar la variable `numTotalPersonal`. Esta variable deberá incrementar su valor en una unidad cada vez que se cree una nueva persona en el programa (es decir, se registren en el hospital). Todas estas variables y atributos serán visibles sólo en esta clase.
- **Constructores**: Debes declarar un constructor sin parámetros de entrada. Este constructor dará al atributo `nombre` el valor inicial "*anónimo"* y al atributo `nacimiento` el valor *0*. Deberás además declarar otro constructor que reciba el nombre y el año de nacimiento de la nueva persona creada en el programa. Cuando implementes los constructores, no olvides que debes llevar un registro del número total de personal del hospital a medida que sean registrados, utilizando `numTotalPersonal`.
- **Métodos get y set**: Por ahora solo será necesario que implementes el método `getNombre()` y el método `setNombre(String nombre)` que te permitirán acceder y modificar el atributo `nombre`.
- **Método adicional**: Es necesario implementar el método `toString()` que pueda devolver una cadena de texto con toda la información de las personas. Sigue el siguiente formato:

```
Nombre: <nombre>. Año de nacimiento: <nacimiento>.
```



Escribe aquí el código de la clase `Persona`:

Apartado 2 (2 puntos)

En este apartado implementarás la clase `Doctor`.

Una vez creada la clase `Persona`, ya puedes crear otras clases que modelen al personal del hospital de forma más específica: doctores, enfermeros, personal de limpieza o mantenimiento, etc. Recuerda que todo el personal del hospital está formado por personas, por lo que debes tener en cuenta lo programado en el apartado anterior para éste. Tras el reparto de tareas en tu equipo de programadores, te encomiendan la programación de la clase `Doctor`. La clase `Doctor` debe:

- Aprovechar lo ya implementado en la clase `Persona`, aplicando los conceptos dados en clase.
- **Variables y atributos**:
  - Debes guardar el número de pacientes que cada doctor tiene a su cargo (`numPacientes`). `numPacientes` sólo debe ser visible en esta clase.
  - Debes guardar la `especialidad` de cada doctor. La `especialidad` debe ser visible sólo en esta clase. Además, el atributo `especialidad` será modelado mediante las siguientes constantes:

    MED_GENERAL = 1; CARDIOLOGIA = 2; PEDIATRIA = 3; OTROS = 4

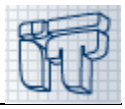
    Por defecto, la especialidad será de tipo OTROS.
- **Constructores**: Te pedimos que implementes un sólo constructor para la clase `Doctor` que reciba como parámetros: el nombre, el año de nacimiento, el número de pacientes que el doctor tiene a su cargo y la especialidad del doctor. Además, este constructor deberá verificar que la especialidad es válida. Un amigo vino en tu ayuda y ya implementó el método:

  ```
  private static boolean especialidadValida(/*debes saber el tipo*/ especialidad)
  ```

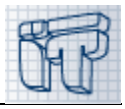
  Tú solo tienes que llamarlo para hacer la verificación. El método `especialidadValida` te devolverá `true` si la especialidad que pasas como parámetro es válida y `false` si la especialidad que le pasas es inválida. Si la especialidad resultase inválida deberás guardar el valor por defecto y, además, mostrar por pantalla el siguiente mensaje de error:

  ```
  Especialidad inválida: se establece OTROS en su lugar
  ```
- **Método adicional**: Es necesario implementar el método `toString()` que pueda devolver una cadena de texto con toda la información de las personas. Debe seguir el siguiente formato:

```
Nombre: <nombre>. Año de nacimiento: <nacimiento>. Especialidad: <especialidad>.
```



Completa el siguiente espacio programando la clase `Doctor`, de forma que contenga toda la información presentada hasta este momento:



Apartado 3 (2 puntos)

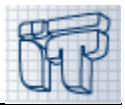
En este apartado será necesario cambiar y/o añadir elementos a las clases `Persona` y `Doctor`.

El reparto de instrumental necesario para doctores, enfermeros y estudiantes en prácticas sigue una organización fija. Todos deben ir al mostrador de la planta en la que estén asignados al comenzar su turno y pedir el instrumental que les corresponda. Sin embargo, para cada tipo específico de personal, el instrumental será diferente.

Para incluir de forma organizada en tu código esto, decides añadir un método llamado `obtenerInstrumental()` en la clase `Persona`. Este método deberá devolver un String con el instrumental entregado. Sin embargo, la clase `Persona` no tiene la información suficiente para implementar el método `obtenerInstrumental()` ya que el instrumental a entregar depende del tipo de personal del Hospital.

Indica cuál sería ahora la declaración de la clase `Persona`, en caso de que hubiera algún cambio en ella. Si no lo hay, repite la declaración de la clase que ya propusiste en el primer apartado.

La clase `Persona` no puede implementar el método `obtenerInstrumental()`, pero las clases que hereden de `Persona`, sí deben hacerlo. ¿Qué debes incluir en la clase `Persona` para que esto sea posible?

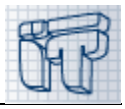


Ahora habrá que realizar modificaciones en la clase `Doctor`.

Como programador de la clase `Doctor`, te han pedido que seas tú quien incluya el método para obtener instrumental en esa clase. Los doctores reciben siempre un estetoscopio, pero la calidad de éste depende de la `especialidad` del doctor. Aquellos doctores que pertenecen a cardiología o pediatría reciben un estetoscopio de alta calidad. Los doctores de medicina general reciben uno de calidad media; y los doctores de otras especialidades, uno de calidad baja.

Implementa el método `obtenerInstrumental()` para la clase `Doctor`, de forma que devuelva:

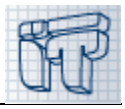
- "`Estetoscopio de calidad alta`" para cardiólogos y pediatras.
- "`Estetoscopio de calidad media`" para médicos generales.
- "`Estetoscopio de calidad baja`" para otros médicos.



Por otra parte, el personal del hospital tiene una serie de habilidades según su puesto de trabajo. Los doctores, por ejemplo, pueden operar pero no extraen sangre, ya que de ello se encargan los enfermeros. Sin embargo, los empleados de limpieza del hospital no pueden hacer ninguna de las dos cosas. Para incluir esta información, se decide hacer uso de la interfaz `Habilidades` que incluye los métodos `puedeOperar()` y `puedeExtraerSangre()`. Ambos métodos devuelven verdadero si esa persona puede realizar esa tarea, y falso en caso contrario. Escribe a continuación la **interfaz** `Habilidades`:

La estructura de clases ya creada debe hacer uso de esa interfaz. Indica cuál sería la declaración de la clase `Persona` en ese caso, partiendo de la declaración que hiciste al comienzo de este apartado. Si no es necesario hacer ningún cambio, copia la declaración de ésta que indicaste al comienzo de este apartado.

El Hospital por ahora no te pide que implementes el resto de la aplicación. Gracias por tu trabajo.



Problema 2 (1 punto)

El Ministerio de Sanidad, Servicios Sociales e Igualdad he decidido aumentar el presupuesto de los hospitales públicos teniendo en cuenta la cantidad de personal que estos tengan. ¡Menos mal que nuestra aplicación guarda este valor en la variable `numTotalPersonal` de la clase `Persona`! El Ministerio dice que el bono que otorgará será:

- 50000 euros si la cantidad de personal es menor a 500 personas
- 100000 euros si la cantidad de personal del Hospital es mayor o igual a 500 personas pero menor de 1500
- 200000 euros si el hospital tiene 1500 o más personas

Como no hay tiempo que perder y este es un asunto muy delicado, el Hospital decidió que un amigo tuyo implementara el método:

```
public static int calculaBono(int cantidadPersonal)
```

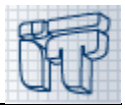
y lo incluyese dentro de la clase `Persona`. Como tú eres experto en Pruebas de Programas, el Hospital te ha asignado las tareas de:

- Diseñar las pruebas a realizar (Apartado 1).
- Hacer un programa en JUnit para probar el programa que tu amigo está haciendo (Apartado 2).

Apartado 1 (0,4 puntos)

Utiliza la técnica de identificación de clases de equivalencia para rellenar la siguiente tabla (define tantas filas como necesites). **Nota**: obviamente la última columna **no** podrás rellenarla.

Identificación (o nombre) de la prueba	Datos(s) entrada	Salida(s) esperada	Salida obtenida



Apartado 2 (0,6 puntos)

Te han pedido que hagas un conjunto de pruebas que busque posibles fallos del método `calculaBono` utilizando la técnica de clases de equivalencia. Por ello, desarrolla un método con JUnit que pruebe todas las clases de equivalencia que has identificado anteriormente.

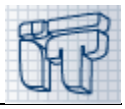
NOTA: Algunas anotaciones y métodos de JUnit que pueden serte útil son:

- *Anotaciones: @Before; @After*
- *@Test método a ejecutar*
- *assertEquals(long esperado, long actual)*

  o si prefieres:

  assertEquals(long actual, long esperado)

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

public class CalculaBonoTest {









}
```



SOLUCIONES DE REFERENCIA (Varias soluciones a cada uno de los problemas son posibles)

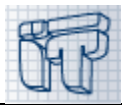
PROBLEMA 1

Apartado 1 (2 puntos)

```
public class Persona {

	private String nombre;
	private int nacimiento;
	private static int numTotalPersonal;

	public Persona() {
		this.nombre = "anónimo";
		this.nacimiento = 0;
		numTotalPersonal++;
	}

/* Segunda opción para el constructor sin argumentos de entrada:
	public Persona() {
		this("anónimo", 0);
	}
*/

	public Persona(String nombre, int nacimiento) {
		this.nombre = nombre;
		this.nacimiento = nacimiento;
		numTotalPersonal++;
	}

	public String getNombre() {
		return this.nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String toString() {
		return "Nombre: " + this.nombre +
				". Año de nacimiento: " + this.nacimiento + ".";
	}
}
```

Apartado 2 (2 puntos)

```
public class Doctor extends Persona {

	private int especialidad;
	private int numPacientes;

	public final static int MED_GENERAL = 1;
	public final static int CARDIOLOGIA = 2;
	public final static int PEDIATRIA = 3;
	public final static int OTROS = 4;
```



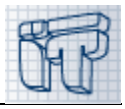
```
	public Doctor(String nombre, int nacimiento, int numPacientes,
			int especialidad) {
		super(nombre, nacimiento);
		this.numPacientes = numPacientes;
		if (especialidadValida(especialidad)) {
			this.especialidad = especialidad;
		} else {
			this.especialidad = OTROS;
			System.err.println("Especialidad inválida: se establece OTROS en
					su lugar");
		}
	}

	public String toString() {
		return super.toString() + "Especialidad: " + this.especialidad + ".";
	}
```

Apartado 3 (2 puntos)

```
### Primer bloque ###

public abstract class Persona

### Segundo bloque ###

public abstract String obtenerInstrumental();

### Tercer bloque ###

public String obtenerInstrumental() {
	if (this.especialidad == CARDIOLOGIA || this.especialidad == PEDIATRIA){
		return "Estetoscopio de calidad alta";
	} else if (this.especialidad == MED_GENERAL) {
		return "Estetoscopio de calidad media";
	} else {
		return "Estetoscopio de calidad baja";
	}
}

### Cuarto bloque ###

public interface Habilidades {

	boolean puedeOperar();
	boolean puedeExtraerSangre();
}

### Quinto bloque ###

public abstract class Persona implements Habilidades
```

Apartado 1 (2 puntos): Persona

- 0,5: Atributos
  - 0,2 Atributos nombre y nacimiento con visibilidad correcta (0,1 para cada uno)
  - 0,3 Atributo estático numTotalPersonas con visibilidad correcta
- 0,3: Constructor sin parámetros de entrada
  - 0,2 Primera línea del constructor y asignación de valores correctos a los atributos
  - 0,1 Incrementar la variable estática numTotalPersonas



- 0,4: Constructor con parámetros de entrada
  - 0,3 Primera línea del constructor y asignación de valores correctos a los atributos
  - 0,1 Incrementar la variable estática numTotalPersonas
- 0,25: Método getNombre()
- 0,25: Método setNombre(String nombre)
- 0,3: Método toString( )
  - 0,2 Crear la String con el formato correcto
  - 0,1 Return
- Los errores significativos están sujetos a sanciones adicionales

Apartado 2 (2 puntos): Doctor

- 0,2: Declaración de clase que extiende la clase Persona
- 0,8: Atributos y constantes
  - 0,2 Atributos especialidad y numPacientes con visibilidad correcta (0,1 para cada uno)
  - 0,6 Constantes referidas a la especialidad
- 0,7: Constructor con parámetros de entrada
  - 0,2 Llamada al constructor padre mediante super
  - 0,1 Asignación del atributo numPacientes
  - 0,4 Comprobación (mediante el método especialidadValida) y asignación del valor de especialidad
    - 0,1 Llamada al método especialidadValida y asignación de la especialidad
    - 0,2 Asignación del valor OTROS
    - 0,1 Mensaje de error
- 0,3: Método toString() que devuelve un String
  - 0,2 Uso del super()
  - 0,1 Crear la String con el formato correcto y return
- Los errores significativos están sujetos a sanciones adicionales

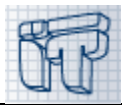
Apartado 3 (2 puntos): Clases abstractas e interfaces

- 0,3: Declaración de clase Persona abstracta
- 0,3: Declaración de método abstracto obtenerInstrumental() en Persona
- 0,5: Método obtenerInstrumental() en Doctor
- 0,5: Interfaz Habilidades
- 0,4: Declaración de clase Persona abstracta que implementa Habilidades
- Los errores significativos están sujetos a sanciones adicionales

PROBLEMA 2

Apartado 1 (0,4 puntos)

Identificación (o nombre) de la prueba	Datos(s) entrada	Salida(s) esperada	Salida obtenida
personalBajo	**100** /* válido cualquier nº ∈ [0, 500) */	50000	
personalMedio	**1000** /* válido cualquier nº ∈ [500, 1500) */	100000	
personalAlto	**2000** /* válido cualquier nº ∈ [1500, ∞) */	200000	



Apartado 2 (0,6 puntos)

Se debe probar un valor incluido en cada rango mencionado en el apartado anterior. Un ejemplo es:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculaBonoTest {

    @Test
    public void calculaBonoPersonalBajoTest() {
       assertEquals(50000, Persona.calculaBono(100)); // Clase de equivalencia 1
    }

    @Test
    public void calculaBonoPersonalMedioTest() {
       assertEquals(100000, Persona.calculaBono(1000)); // Clase de equivalencia 2
    }

    @Test
    public void calculaBonoPersonalAltoTest() {
       assertEquals(200000, Persona.calculaBono(2000)); // Clase de equivalencia 3
    }

}
```

Apartado 1 (0,4 puntos):

- 0,4: Identificación correcta de las clases
  - 0,1 No colocar más pruebas de las necesarias
  - 0,3 Pruebas correctas (0,1 por prueba)

Apartado 2 (0,6 puntos):

- En caso de hacer un solo método @Test:
  - 0,1: Declaración del método marcado como @Test
  - 0,5: Creación correcta del test assert
- En caso de hacer 3 métodos @Test, para cada uno:
  - 0,1: Declaración del método marcado como @Test
  - 0,1: Creación correcta del test assert