



Grado en Ingeniería Informática
Estructura de Datos y Algoritmos, Grupo 80M, 2014/2015
09 de Marzo de 2015

Nombre y Apellidos:

.....

PROBLEMA 1 (1 punto) – Programación Orientada a Objetos.

Una compañía solicita el desarrollo de una aplicación para gestionar los datos de sus empleados.

Hay dos clases de empleados. Los empleados asalariados que cobran un salario neto mensual y los empleados autónomos que cobran por el número de horas trabajadas en el mes. Los datos a almacenar para un empleado son el siguiente:

- DNI del empleado
- Nombre del empleado
- Lista de los cursos: Array de String.
- Número total de los cursos
- En el caso de un empleado asalariado, se necesita guardar el salario mensual bruto.
- En el caso de un empleado autónomo, se necesita guardar el número total de las horas trabajadas en el mes, y el precio bruto de cada hora.

Cada vez que un empleado (asalariado o autónomo) se registra a un curso, se comprueba primero si dicho curso está en la lista de cursos realizados por el empleado.

Si el curso está en la lista no se registrará el nuevo curso y se informará al empleado por mensaje que *no se admite registrarse al mismo curso*. Si el curso no está, el nombre del curso se añade a la lista almacenada para cada empleado. Además, si el empleado ya ha alcanzado el número máximo de cursos permitido (3 cursos), no se registrará el nuevo curso y se informará por mensaje que *se ha alcanzado el máximo número de cursos permitidos*.

Los empleados asalariados que participan en los cursos tienen derecho de una desgravación del IRPF. Es decir, por ejemplo, si el salario bruto mensual de un empleado es 2000 €, IRPF=0.2 y ha participado en 3 cursos. El salario neto será $= 2000 * (1 - (0.2 - 0.03)) = 1660$ €.

Mientras el empleado autónomo no tiene este derecho, es decir, por ejemplo, si un empleado ha hecho 120 horas en el mes, y el precio de una hora es 10 €. El salario neto será $(120 \cdot 10) \cdot (1 - 0.2) = 960$ €.

Nota: IRPF=0.2 (20%), es un porcentaje fijo de impuesto que se aplica igual a todos los empleados.

Se pide desarrollar la(s) estructura(s) de datos necesaria(s) para implementar la solución. En concreto, será necesario:

- método(s) constructor(es) necesarios con sus parámetros para crear objetos de las clases.
- esRepetido-> recibe como parámetro el nombre del curso al que quiere apuntar el empleado. Devuelve true si el empleado está apuntado a dicho curso y false en otro caso.
- inscribirCurso-> recibe como parámetro el nombre del curso y lo guarda en la lista de cursos del empleado. Este método debería comprobar el número de los cursos que no pasa el límite permitido, y también utiliza el método anterior esRepetido para comprobar si el empleado está apuntado o no al curso en cuestión.
- calcularSalario-> devuelve el salario neto (tipo float) del empleado.

La solución propuesta debe estar diseñada bajo los principios de la programación orientada a objetos, que permitan obtener software robusto, reutilizable y fácil de adaptar.

Notas:

- No es necesario implementar los métodos getters and setters.
- No es necesario implementar un método main o una clase Test para probar la aplicación, es suficiente con diseñar la(s) clase(s).
- Para simplificar el problema, vamos a suponer que los cursos no se pueden eliminar de la lista de cursos del empleado.

Solución:

```
public abstract class Empleado {
    String dni;
    String nombre;
    String [] lstCursos;
    int numCursos;
    static float iprf = 0.2f;

    public Empleado(String dni, String nombre) {
        this.dni = dni;
        this.nombre = nombre;
        this.lstCursos = new String [3];
    }

    public boolean esRepetido (String curso){
        for (int i=0; i<lstCursos.length; i++)
            if (lstCursos[i].equalsIgnoreCase(curso)) return true;
        return false;
    }

    public void inscribirCurso (String curso){
        if (esRepetido(curso)){
```

```

        System.out.println("no se admite registrarse al mismo curso");
    }else{
        if (lstCursos.length==numCursos){
            System.out.println("se ha alcanzado el máximo número de cursos
permitidos");
        }else{
            lstCursos[numCursos]=curso;
            numCursos++;
        }
    }
}

public abstract float calculrSalarioNeto();

}

public class Asariado extends Empleado {

    float salarioBruto;

    public Asariado(String dni, String nombre, float salarioBruto) {
        super(dni, nombre);
        this.salarioBruto = salarioBruto;
        // TODO Auto-generated constructor stub
    }

    public float calculrSalarioNeto(){
        return (float) (salarioBruto * (1-(iprf-((float)numCursos)*0.01)));
    }
}

public class Autonomo extends Empleado {

    int horas;
    float precioHora;
    public Autonomo(String dni, String nombre, int horas, float precioHora) {
        super(dni, nombre);
        this.horas=horas;
        this.precioHora=precioHora;

        // TODO Auto-generated constructor stub
    }

    public float calculrSalarioNeto(){
        return (float) (horas*precioHora * (1-(iprf)));
    }
}

```

PROBLEMA 2 (1 punto)- Dada las siguientes clases que implementan una lista doblemente enlazada:

```
public class DNode<E> {
    public E element;
    public DNode<E> prev, next;

    public DNode(E objeto){
        element = objeto;
    }
}

public class DList<E> {
    /**Nodo centinela cuyo propiedad next apunta al primer nodo de la
    lista*/
    public DNode<E> header;
    /**Nodo centinela cuyo propiedad prev apunta al último nodo de la
    lista*/
    public DNode<E> tailer;

    .....

    public boolean isEmpty() {...}
    public void addFirst(E e) {...}
    public void addLast(E e) {...}
    public void addBefore(E e, E n) {...}
    public void addAfter(E e, E n) {...}

}
```

Dado el fragmento de código anterior, completa el siguiente método que reciba un objeto **Integer** y que inserte ese objeto en una *lista doblemente enlazada ordenada* de una manera ascendente.

Se pide implementar el método insertInOrden.

Solución:

```

public void insertInOrden(Integer e) {
    DNode<Integer> newNode=new DNode<Integer>(e);
    if (isEmpty()) {
        addFirst(e);
        return;
    } else {
        DNode<Integer> aux=header.next;

        //Recorremos la lista hasta encontrar la posición
        while (aux!=tailer && aux.element<newNode.element) {
            aux=aux.next;
        }
        if (aux==tailer) {
            //Hemos llegado al final de la lista
            addLast(e);
        } else {
            addBefore(aux.element, newNode.element);
        }
    }
}

```