7.G. Anexos de ejercicios resueltos.

Tabla de contenidos

1 Anexo I.- Elaboración de los constructores de la clase Rectangulo.

ENUNCIADO

Intenta rescribir los constructores de la clase `Rectangulo` teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, **x1**, **y1**, **x2**, **y2**, que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros, **punto1**, **punto2**, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, **base** y **altura**, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

POSIBLE SOLUCIÓN

Durante el proceso de creación de un objeto (**constructor**) de la **clase contenedora** (en este caso **Rectangulo**) hay que tener en cuenta también la creación (llamada a **constructores**) de aquellos objetos que son contenidos (en este caso objetos de la clase `Punto`).

En el caso del primer **constructor**, habrá que crear dos **puntos** con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (`vertice1` y `vertice2`):

```
public Rectangulo ()
```

```
{
```

```
    this.vertice1= new Punto (0,0);
```

```
    this.vertice2= new Punto (1,1);
```

```
}
```

Para el segundo **constructor** habrá que crear dos puntos con las coordenadas `x1` , `y1` , `x2` , `y2` que han sido pasadas como parámetros:

```
public Rectangulo (double x1, double y1, double x2, double y2)
```

```
{
```

```
    this.vertice1= new Punto (x1, y1);
```

```
    this.vertice2= new Punto (x2, y2);
```

```
}
```

En el caso del tercer **constructor** puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo:

Ahora bien, esto podría ocasionar un **efecto colateral** no deseado si esos objetos de tipo `Punto` son modificados en el futuro desde el código cliente del **constructor** (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizá fuera recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al **constructor** de la clase `Punto` con los valores de los atributos (x, y).
2. Llamar al **constructor copia** de la clase `Punto`, si es que se dispone de él.

Aquí tienes las dos posibles versiones:

Constructor que "extrae" los atributos de los parámetros y crea nuevos objetos:

```
public Rectangulo (Punto vertice1, Punto vertice2)
```

```
{
```

```
    this.vertice1= vertice1;
```

```
    this.vertice2= vertice2;
```

```
}
```

Constructor que crea los nuevos objetos mediante el **constructor copia** de los parámetros:

```
public Rectangulo (Punto vertice1, Punto vertice2)
```

```
{
```

```
    this.vertice1= new Punto (vertice1.obtenerX(), vertice1.obtenerY() );
```

```
    this.vertice2= new Punto (vertice2.obtenerX(), vertice2.obtenerY() );
```

```
}
```

En este segundo caso puedes observar la utilidad de los **constructores de copia** a la hora de tener que **clonar** objetos (algo muy habitual en las inicializaciones).

Para el caso del **constructor** que recibe como parámetros la base y la altura, habrá que crear sendos vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

```
public Rectangulo (Punto vertice1, Punto vertice2)
```

```
{
```

```
     this.vertice1= new Punto (vertice1 );
```

```
     this.vertice2= new Punto (vertice2 );
```

```
}
```

Quedaría finalmente por implementar el **constructor copia**:

```
// Constructor copia
```

```
public Rectangulo (Rectangulo r) {
```

```
    this.vertice1= new Punto (r.obtenerVertice1() );
```

```
    this.vertice2= new Punto (r.obtenerVertice2() );
```

```
}
```

En este caso nuevamente volvemos a **clonar** los atributos `vertice1` y `vertice2` del objeto **r** que se ha pasado como parámetro para evitar tener que compartir esos atributos en los dos rectángulos.

2 Anexo II.- Métodos para las clases heredadas Alumno y Profesor.

ENUNCIADO

Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos **get** y **set** en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos).

POSIBLE SOLUCIÓN

1. **Clase `Alumno`.**

Se trata de heredar de la clase `Persona` y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```
public class Alumno extends Persona {
```

```
        protected String grupo;
```

```
        protected double notaMedia;
```

```

```

```
        // Método getNombre
```

```
        public String getNombre (){
```

```
            return nombre;
```

```
        }
```

```

```

```
        // Método getApellidos
```

```
        public String getApellidos (){
```

```
            return apellidos;
```

```
        }
```

```

```

```
        // Método getFechaNacim

        public GregorianCalendar getFechaNacim (){

            return this.fechaNacim;

        }


        // Método getGrupo

        public String getGrupo (){

            return grupo;

        }


        // Método getNotaMedia

        public double getNotaMedia (){

            return notaMedia;

        }


        // Método setNombre

        public void setNombre (String nombre){

            this.nombre= nombre;

        }


        // Método setApellidos
```

```
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }

        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }

        // Método setGrupo
        public void setGrupo (String grupo){
            this.grupo= grupo;
        }

        // Método setNotaMedia
        public void setNotaMedia (double notaMedia){
            this.notaMedia= notaMedia;
        }

}
```

Si te fijas, puedes utilizar sin problema la referencia `this` a la propia clase con esos atributos heredados, pues pertenecen a la clase: `this.nombre`, `this.apellidos`, etc.

2. **Clase `Profesor`.**

Seguimos exactamente el mismo procedimiento que con la clase `Alumno`.

```
public class Profesor extends Profesor {
```

```
        String especialidad;
```

```
        double salario;
```

```

```

```
        // Método getNombre
```

```
        public String getNombre (){
```

```
            return nombre;
```

```
        }
```

```

```

```
        // Método getApellidos
```

```
        public String getApellidos (){
```

```
            return apellidos;
```

```
        }
```

```

```

```
        // Método getFechaNacim
```

```
        public GregorianCalendar getFechaNacim (){
```

```
            return this.fechaNacim;
```

```
        }
```

```

```

```
        // Método getEspecialidad
```

```
    public String getEspecialidad (){
        return especialidad;
    }

    // Método getSalario
    public double getSalario (){
        return salario;
    }

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

    // Método setApellidos
    public void setApellidos (String apellidos){
        this.apellidos= apellidos;
    }

    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
```

```
            this.fechaNacim= fechaNacim;
        }

        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
        }

        // Método setESpecialidad
        public void setESpecialidad (String especialidad){
            this.especialidad= especialidad;
        }
}
```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos **get** y **set** para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase `Alumno` y otros seis en la clase `Profesor`. Así que recuerda: **se pueden heredar tanto los atributos como los métodos**.

Aquí tienes un ejemplo de cómo podrías haber definido la clase `Persona` para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```
public class Persona {
        protected String nombre;
        protected String apellidos;
        protected GregorianCalendar fechaNacim;
```

```
        // Método getNombre
        public String getNombre (){
            return nombre;
        }

        // Método getApellidos
        public String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }

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

```
        // Método setApellidos

        public void setApellidos (String apellidos){

            this.apellidos= apellidos;

        }


        // Método setFechaNacim

        public void setFechaNacim (GregorianCalendar fechaNacim){

            this.fechaNacim= fechaNacim;

        }


}
```

3 Anexo III.- Métodos para los atributos de las clases Alumno y Profesor.

ENUNCIADO

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos **get** y **set** en la clase `Persona` para trabajar con sus tres atributos y en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para `Persona` van a ser heredados en `Alumno` y en `Profesor`.

POSIBLE SOLUCIÓN

1. **Clase `Persona`.**

```
public class Persona {
```

```
        protected String nombre;
```

```
        protected String apellidos;
```

```
        protected GregorianCalendar fechaNacim;
```

```

```

```
        // Método getNombre
```

```
        public String getNombre (){
```

```
            return nombre;
```

```
        }
```

```

```

```
        // Método getApellidos
```

```
        public String getApellidos (){
```

```
            return apellidos;
```

```
        }
```

```

```

```
        // Método getFechaNacim

        public GregorianCalendar getFechaNacim (){

            return this.fechaNacim;

        }


        // Método setNombre

        public void setNombre (String nombre){

            this.nombre= nombre;

        }


        // Método setApellidos

        public void setApellidos (String apellidos){

            this.apellidos= apellidos;

        }


        // Método setFechaNacim

        public void setFechaNacim (GregorianCalendar fechaNacim){

            this.fechaNacim= fechaNacim;

        }
```

2. **Clase `Alumno`.**

Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado.

```
public class Alumno extends Persona {
```

```
        protected String grupo;
```

```
        protected double notaMedia;
```

```

```

```
        // Método getGrupo
```

```
        public String getGrupo (){
```

```
            return grupo;
```

```
        }
```

```

```

```
        // Método getNotaMedia
```

```
        public double getNotaMedia (){
```

```
            return notaMedia;
```

```
        }
```

```

```

```
        // Método setGrupo
```

```
        public void setGrupo (String grupo){
```

```
            this.grupo= grupo;
```

```
        }
```

```

```

```
        // Método setNotaMedia
```

```
        public void setNotaMedia (double notaMedia){
```

```
            this.notaMedia= notaMedia;
```

```
        }
```

```

```

```
}
```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

3. Clase `Profesor`.

Seguimos exactamente el mismo procedimiento que con la clase `Alumno`.

```
public class Profesor extends Profesor {
```

```
        String especialidad;
```

```
        double salario;
```

```

```

```
        // Método getEspecialidad
```

```
        public String getEspecialidad (){
```

```
            return especialidad;
```

```
        }
```

```

```

```
        // Método getSalario
```

```
        public double getSalario (){
```

```
            return salario;
```

```
        }

        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
        }

        // Método setEspecialidad
        public void setESpecialidad (String especialidad){
            this.especialidad= especialidad;
        }
}
```

4 Anexo IV.- Contextos del modificador final.

Distintos contextos en los que puede aparecer el modificador final	
Lugar	**Función**
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método.	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Veamos un ejemplo de cada posibilidad:

1. **Modificador de una clase.**

```
public final class ClaseSinDescendencia {     // Clase “no heredable”
```

```
    …
```

```
}
```

2. **Modificador de un atributo.**

```
public class ClaseEjemplo {
```

```
  // Valor constante conocido en tiempo de compilación
```

```
  final double PI= 3.14159265;
```

```

```

```
   // Valor constante conocido solamente en tiempo de ejecución

   final int SEMILLA= (int) Math.random()*10+1;

   …

}
```

3. **Modificador de un método**.

```
public final metodoNoRedefinible (int parametro1) {   // Método “no redefinible”

     …

}
```

4. **Modificador en una variable referencia.**

```
// Referencia constante: siempre se apuntará al mismo objeto Alumno recién creado,

// aunque este  objeto pueda sufrir modificaciones.

final Alumno PRIMER_ALUMNO= new Alumno (“Pepe”, “Torres”, 9.55);   // Ref. constante


// Si la variable no es una referencia (tipo primitivo), sería una constante más

// (como un atributo constante).

final int NUMERO_DIEZ= 10;     // Valor constante (dentro del ámbito de vida de la variable)
```

5. **Modificador en un parámetro de un método.**

```
void metodoConParametrosFijos (final int par1, final int par2) {

     // Los parámetros “par1” y “par2” no podrán sufrir modificaciones aquí dentro

     …
```

```
}
```