7.B. Composición.

Sitio: Formación Profesional a Distancia
Curso: Programación
Libro: 7.B. Composición.
Imprimido por: Iván Jiménez Utiel
Día: lunes, 10 de febrero de 2020, 15:48

Tabla de contenidos

1. Composición.

¿Cómo hay que hacer para establecer una relación de composición? ¿Es necesario indicar algún modificador al definir las clases? En tal caso, ¿se indicaría en la clase continente o en la contenida? ¿Afecta de alguna manera al código que hay que escribir? En definitiva, ¿cómo se indica que una clase contiene instancias de otra clase en su interior?

1.1. Sintaxis de la composición.

Para indicar que una clase contiene objetos de otra clase no es necesaria **ninguna sintaxis especial**. Cada uno de esos objetos no es más que un atributo y, por tanto, debe ser declarado como tal:

```
class <nombreClase> {
    [modificadores] <NombreClase1> nombreAtributo1;
    [modificadores] <NombreClase2> nombreAtributo2;
…
}
```

En unidades anteriores has trabajado con la clase **`Punto`**, que definía las coordenadas de un punto en el plano, y con la clase **`Rectangulo`**, que definía una figura de tipo rectángulo también en el plano a partir de dos de sus **vértices** (**inferior izquierdo** y **superior derecho**). Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de **composición**: "**un rectángulo contiene puntos**". Por tanto, podrías ahora redefinir los atributos de la clase **`Rectangulo`** (cuatro **números reales**) como dos objetos de tipo **`Punto`**:

```
class Rectangulo {
    private Punto vertice1;
    private Punto vertice2;
…
}
```

Ahora los métodos de esta clase deberán tener en cuenta que ya no hay cuatro atributos de tipo **`double`**, sino dos atributos de tipo **`Punto`** (cada uno de los cuales contendrá en su interior dos atributos de tipo **`double`**).

Autoevaluación

Para declarar un objeto de una clase determinada, como atributo de otra clase, es necesario especificar que existe una relación de composición entre ambas clases mediante el modificador object**. ¿Verdadero o Falso?**

- Verdadero
- Falso

Ejercicio resuelto

Intenta rescribir los siguientes los métodos 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. **Método `calcularSuperfice`, que calcula y devuelve el área de la superficie encerrada por la figura.**
2. **Método `calcularPerimetro`, que calcula y devuelve la longitud del perímetro de la figura.**

Solución:

En ambos casos la **interfaz** no se ve modificada en absoluto (desde fuera su funcionamiento es el mismo), pero internamente deberás tener en cuenta que ya no existen los atributos `x1`, `y1`, `x2`, `y2`, de tipo `double`, sino los atributos `vertice1` y `vertice2` de tipo `Punto`.

```
public double calcularSuperficie () {
    double area, base, altura;   // Variables locales

    base= vertice2.obtenerX () - vertice1.obtenerX ();  // Antes era x2 - x1
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1
    area= base * altura;
    return area;
}

public double CalcularPerimetro () {
    double perimetro, base, altura;   // Variables locales

    base= vertice2.obtenerX () - vertice1.obtenerX ();  // Antes era x2 - x1
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1
    perimetro= 2*base + 2*altura;
    return perimetro;
}
```

1.2. Uso de la composición (I). Preservación de la ocultación.

Como ya has observado, la relación de **composición** no tiene más misterio a la hora de implementarse que simplemente declarar **atributos** de las clases que necesites dentro de la clase que estés diseñando.

Ahora bien, cuando escribas clases que contienen objetos de otras clases (lo cual será lo más habitual) deberás tener un poco de precaución con aquellos métodos que devuelvan información acerca de los **atributos** de la clase (métodos "**obtenedores**" o de tipo **get**).

Como ya viste en la unidad dedicada a la creación de clases, lo normal suele ser declarar los **atributos** como **privados** (o **protegidos**, como veremos un poco más adelante) para ocultarlos a los posibles **clientes** de la clase (otros objetos que en el futuro harán uso de la clase). Para que otros objetos puedan acceder a la información contenida en los **atributos,** o al menos a una parte de ella, deberán hacerlo a través de **métodos que sirvan de interfaz**, de manera que sólo se podrá tener acceso a aquella información que el creador de la clase haya considerado oportuna. Del mismo modo, los **atributos** solamente serán modificados desde los métodos de la clase, que decidirán cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la **interfaz** con el exterior.

Hasta ahora los métodos de tipo **get** devolvían **tipos primitivos**, es decir, copias del contenido (a veces con algún tipo de modificación o de formato) que había almacenado en los **atributos**, pero los atributos seguían "a salvo" como elementos privados de la clase. Pero, a partir de este momento, al tener objetos dentro de las clases y no sólo tipos primitivos, es posible que en un determinado momento interese devolver un **objeto completo**.

Ahora bien, cuando vayas a devolver un objeto habrás de obrar con mucha precaución. Si en un método de la clase devuelves directamente un objeto que es un atributo, estarás ofreciendo directamente una **referencia** a un objeto atributo que probablemente has definido como privado. ¡De esta forma estás **volviendo a hacer público un atributo que inicialmente era privado**!

Para evitar ese tipo de situaciones (ofrecer al exterior referencias a objetos privados) puedes optar por diversas alternativas, procurando siempre evitar la devolución directa de un atributo que sea un objeto:

- Una opción podría ser devolver siempre tipos primitivos.

- Dado que esto no siempre es posible, o como mínimo poco práctico, otra posibilidad es crear un nuevo objeto que sea una copia del atributo que quieres devolver y utilizar ese objeto como valor de retorno. Es decir, **crear una copia del objeto** especialmente para devolverlo. De esta manera, el código cliente de ese método podrá manipular a su antojo ese nuevo objeto, pues no será una referencia al atributo original, sino un nuevo objeto con el mismo contenido.

Por último, debes tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos). En tales casos, no habrá problema en devolver directamente el atributo para que el código llamante (cliente) haga el uso que estime oportuno de él.

Debes evitar por todos los medios la devolución de un atributo que sea un objeto (estarías dando directamente una referencia al atributo, visible y manipulable desde fuera), salvo que se trate de un caso en el que deba ser

así.

Para entender estas situaciones un poco mejor, podemos volver al objeto `Rectangulo` y observar sus nuevos métodos de tipo **get**.

Ejercicio resuelto

Dada la clase `Rectangulo`, escribe sus nuevos métodos `obtenerVertice1` y `obtenerVertice2` para que devuelvan los vértices inferior izquierdo y superior derecho del rectángulo (objetos de tipo `Punto`), teniendo en cuenta su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

Solución:

Los métodos de obtención de vértices devolverán objetos de la clase **Punto**:

```
public Punto obtenerVertice1 ()
{
    return vertice1;
}

public Punto obtenerVertice2 ()
{
    return vertice2;
}
```

Esto funcionaría perfectamente, pero deberías tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera haciendo público un atributo que fue declarado como privado.

Para evitar que esto suceda bastaría con crear un nuevo objeto que fuera una copia del atributo que se desea devolver (en este caso un objeto de la clase **Punto**).

Aquí tienes algunas posibilidades:

```
public Punto obtenerVertice1 ()   // Creación de un nuevo punto extrayendo sus atributos
{
    double x, y;
    Punto p;
    x= vertice1.obtenerX();
```

```
    y= vertice1.obtenerY();
    p= new Punto (x,y);

    return p;
}

public Punto obtenerVertice1 ()  // Utilizando el constructor copia de Punto (si es que está definido)
{
    Punto p;
    p= new Punto (this.vertice1); // Uso del constructor copia
    return p;
}
```

De esta manera, se devuelve un punto totalmente nuevo que podrá ser manipulado sin ningún temor por parte del código cliente de la clase pues es una copia para él.

Para el método **obtenerVertice2** sería exactamente igual.

1.3. Uso de la composición (II). Llamadas a constructores.

Otro factor que debes considerar, a la hora de escribir clases que contengan como atributos objetos de otras clases, es su comportamiento a la hora de instanciarse. Durante el proceso de creación de un objeto (**constructor**) de la clase contenedora habrá que tener en cuenta también la creación (llamadas a **constructores**) de aquellos objetos que son contenidos.

El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

En este caso hay que tener cuidado con las referencias a **objetos que se pasan como parámetros** para rellenar el contenido de los atributos. Es conveniente hacer una copia de esos objetos y utilizar esas copias para los atributos pues si se utiliza la referencia que se ha pasado como parámetro, el código cliente de la clase podría tener acceso a ella sin necesidad de pasar por la interfaz de la clase (volveríamos a dejar abierta una **puerta pública** a algo que quizá sea privado).

Además, si el **objeto parámetro** que se pasó al **constructor** formaba parte de otro objeto, esto podría ocasionar un desagradable efecto colateral si esos objetos son modificados en el futuro desde el código cliente de la clase, ya que no sabes de dónde provienen esos objetos, si fueron creados especialmente para ser usados por el nuevo objeto creado o si pertenecen a otro objeto que podría modificarlos más tarde. Es decir, correrías el riesgo de estar "compartiendo" esos objetos con otras partes del código, sin ningún tipo de control de acceso y con las nefastas consecuencias que eso podría tener: cualquier cambio de ese objeto afectaría a partes del programa supuestamente independientes, que entienden ese objeto como suyo.

En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto, y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.

Recuerda también que sólo se crean objetos cuando se llama a un constructor (uso de `new`). Si realizas asignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino simplemente de las referencias, y por tanto se tratará siempre del mismo objeto.

Se trata de un efecto similar al que sucedía en los métodos de tipo **get**, pero en este caso en sentido contrario (en lugar de que nuestra clase "regale" al exterior uno de sus atributos objeto mediante una referencia, en esta ocasión se "adueña" de un parámetro objeto que probablemente pertenezca a otro objeto y que es posible que el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase `Rectangulo` que contiene en su interior dos objetos de la clase `Punto`. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase `Punto` evitando las referencias a parámetros (haciendo copias).

Autoevaluación

Si se declaran dos variables objeto a y b de la clase X, ambas son instanciadas mediante un constructor, y posteriormente se realiza la asignación a=b**, el contenido de b será una copia del contenido de a, perdiéndose los valores iniciales de b. ¿Verdadero o Falso?**

- Verdadero

Falso

Ejercicio resuelto

Intenta escribir 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.**

Solución:

Esta es una 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.

1.4. Clases anidadas o internas.

En algunos lenguajes, es posible definir una clase dentro de otra clase (**clases internas**):

```
class claseContenedora {
    // Cuerpo de la clase
…
    class claseInterna {
        // Cuerpo de la clase interna
…
    }
}
```

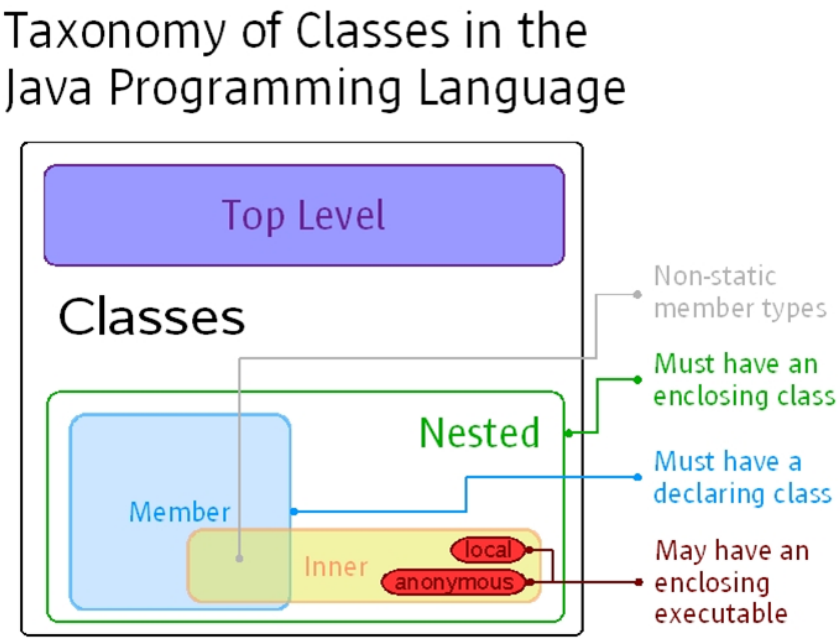

Imagen extraída de curso Programación del MECD.

Se pueden distinguir varios tipos de **clases internas**:

- **Clases internas estáticas** (o **clases anidadas**), declaradas con el modificador `static`.
- **Clases internas miembro**, conocidas habitualmente como **clases internas**. Declaradas al máximo nivel de la clase contenedora y no estáticas.
- **Clases internas locales**, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- **Clases anónimas**, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la **gestión de eventos** en los **interfaces gráficos**.

Aquí tienes algunos ejemplos:

```
class claseContenedora {
…
    static class claseAnidadaEstatica {
…
    }
    class claseInterna {
…
    }
```

Las **clases anidadas**, como miembros de una clase que son (miembros de `claseExterna`), pueden ser declaradas con los modificadores `public`, `protected`, `private` o **de paquete**, como el resto de miembros.

Las **clases internas** (no estáticas) tienen acceso a otros miembros de la clase dentro de la que está definida aunque sean privados (se trata en cierto modo de un miembro más de la clase), mientras que las anidadas (estáticas) no.

Las **clases internas** se utilizan en algunos casos para:

- **Agrupar** clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
- Incrementar el nivel de **encapsulación** y **ocultamiento**.
- Proporcionar un **código fuente más legible y fácil de mantener** (el código de las **clases internas** y **anidadas** está más cerca de donde es usado).

En Java es posible definir **clases internas** y **anidadas**, permitiendo todas esas posibilidades. Aunque para lo ejemplos con los que vas a trabajar no las vas a necesitar por ahora.