«Talento Tech»

Back-End

Java

Clase 06

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase 6: Encapsulamiento y colaboración entre clases

Índice

Objetivos de la clase

- Entender el **encapsulamiento** y las diferentes visibilidades (`public`, `private`, `protected`), garantizando que las clases controlen su estado interno.
- Aprender a usar **getters y setters** para acceder y modificar atributos de forma controlada.
- Explorar la **colaboración entre clases**, creando objetos dentro de otros objetos para modelar relaciones del mundo real.
- Comprender el uso de **variables de clase (estáticas)** y cuándo resultan útiles.
- Aplicar estos conceptos en el contexto de TechLab, logrando un diseño más seguro y coherente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Bienvenida a TechLab

Talento Lab

¡Te damos la bienvenida nuevamente a **TechLab**! Hasta ahora hemos aprendido los fundamentos de POO, creando clases y objetos y comprendiendo su importancia en la organización del código. En esta oportunidad nos centraremos en cómo asegurar que los objetos protejan su información interna (encapsulamiento), cómo interactúan entre sí (colaboración) y cómo aprovechar al máximo las variables de clase. Este es uno de los diferenciales en los productos que ofrece TechLab, ¡por lo que poné atención!

El objetivo es que nuestro código no sólo funcione sino que sea robusto, mantenible y fácil de entender. A medida que el proyecto crece, la comunicación entre objetos y la protección de datos se vuelven esenciales para evitar errores y mantener una arquitectura sólida. ¡Veamos cómo!

Encapsulamiento y visibilidad (public, private, protected)

Encapsulamiento significa ocultar la implementación interna de una clase, exponiendo solo lo necesario a través de una interfaz pública. Esto protege los datos internos de cambios indebidos y hace que el código sea más flexible a la hora de modificarse internamente sin afectar al resto del sistema.

- **public:** El miembro es accesible desde cualquier lugar.
- **private:** El miembro es accesible solo desde dentro de la misma clase.
- **protected:** El miembro es accesible desde la misma clase, subclases y clases del mismo paquete (su comportamiento exacto depende del contexto de paquetes y herencia).

Lo más habitual es que los atributos de una clase sean `private` para evitar que cualquier parte del código los modifique directamente.

Problema: evitar modificaciones inesperadas del estado

Contexto: Antes, el atributo `precio` de `Producto` era público. Cualquier parte del código podía poner `producto.precio = -50.0;` lo cual no tiene sentido. Silvia se queja de estas inconsistencias en los datos y propone resolverlo.

Solución: Declarar el precio como `private` y proporcionar métodos de acceso controlados:

```
public class Producto {
   private String nombre;
   private double precio; // Ahora es privado
   private int cantidadEnStock;
}
```

De este modo, ninguna otra clase puede hacer `producto.precio = x;` directamente. Tendrá que pasar por métodos que validen el valor.

Métodos de acceso (Getters y Setters).

Getters y Setters son métodos que sirven para leer y modificar atributos privados. Un getter retorna el valor de un atributo, mientras que un setter lo modifica, aplicando validaciones si es necesario.

```
public class Producto {
   private String nombre;
   private double precio;
   private int cantidadEnStock;
```

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

    public void setNombre(String nombre) {
        // Podemos verificar que el nombre no esté vacío
        if (nombre != null && !nombre.trim().isEmpty()) {
            this.nombre = nombre;
        }
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        if (precio >= 0) {
            this.precio = precio;
        }
    }

    public int getCantidadEnStock() {
        return cantidadEnStock;
    }

    public void setCantidadEnStock(int cantidadEnStock) {
        if (cantidadEnStock >= 0) {
            this.cantidadEnStock = cantidadEnStock;
        }
    }
}
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Problema: controlar el acceso a los atributos del producto.

Contexto: Silvia exige que jamás se setee un precio negativo. Con los setters, Matías y Sabrina ahora pueden asegurarse de que el precio sea siempre válido al tratar de asignarlo.

Solución: El setter `setPrecio` valida antes de asignar. Si la entrada no es válida, no cambia el atributo, manteniendo la coherencia interna del objeto.

Colaboración entre clases y objetos dentro de objetos.

En un sistema orientado a objetos, estos rara vez están aislados. Suelen colaborar entre sí. La **colaboración** se da cuando un objeto utiliza los métodos o datos de otro objeto para cumplir con su funcionalidad.

A menudo, esta colaboración implica **objetos dentro de objetos**. Por ejemplo, la clase `Pedido` puede tener una lista de `Producto`:

```
public class Pedido {
    private ArrayList<Producto> productos;
    private Cliente cliente;

    public Pedido(Cliente cliente) {
        this.cliente = cliente;
        this.productos = new ArrayList<>();
    }

    public void agregarProducto(Producto p) {
        productos.add(p);
    }
```

```
 public double calcularTotal() {
        double total = 0;
        for (Producto p : productos) {
            total += p.getPrecio() * p.getCantidadEnStock();
        }
        return total;
    }
}
```

Aquí, `Pedido` colabora con `Producto` y `Cliente`. El `Pedido` necesita conocer sus `productos` y el `cliente` asociado y utiliza los getters de `Producto` para obtener precios y calcular el total.

Problema: un pedido con múltiples productos.

Contexto: Silvia pide que cuando se cree un pedido, se pueda asociar varios productos y luego calcular el total. Antes se trabajaba con datos sueltos. Ahora, con POO, `Pedido` es un objeto que tiene `Producto` dentro de él.

Solución: Al crear un `Pedido`, le pasamos un `Cliente` y luego `agregarProducto` para incluir `Producto`. Así, `Pedido` y `Producto` colaboran de forma encapsulada y ordenada.

Variables de clase (static).

Una **variable de clase** (o variable *estática*) es compartida por todas las instancias de la clase. Pertenece a la clase en sí, no a un objeto individual. Esto es útil para llevar contadores globales o configuraciones comunes.

```
public class Pedido {
    private ArrayList<Producto> productos;
    private Cliente cliente;

    public Pedido(Cliente cliente) {
        this.cliente = cliente;
        this.productos = new ArrayList<>();
    }

    public void agregarProducto(Producto p) {
        productos.add(p);
    }

    public double calcularTotal() {
        double total = 0;
        for (Producto p : productos) {
            total += p.getPrecio() * p.getCantidadEnStock();
        }
        return total;
    }
}
```

Cuando creamos `new Producto("Café");` se incrementa `contadorProductos`. Todas las instancias comparten este valor. Podremos consultar la cantidad total de productos creados usando `Producto.getContadorProductos()` sin necesidad de tener un objeto específico.

Problema: contar la cantidad de productos creados.

Contexto: Silvia quiere estadísticas: ¿cuántos productos se han creado en total desde que arrancó el sistema? Esto no es atributo de un producto individual, sino de la clase `Producto` en general.

Solución: Una variable estática `contadorProductos` se incrementa cada vez que se crea un producto. Así, `Producto.getContadorProductos()` retorna la cantidad total creada hasta el momento.

Situación Inicial en TechLab .

Silvia, la Product Owner, ha observado que ciertos datos del sistema se están modificando de manera impredecible. A veces, por error, algún método deja un producto con precio negativo o un cliente con email vacío. Matías y Sabrina entienden que no pueden dejar que cualquier parte del código cambie atributos sin control.

Además, ahora quieren que un **Pedido** (una nueva clase) contenga varios objetos `Producto`. Este escenario requiere que las clases colaboren entre sí, ya que el `Pedido` deberá conocer los productos que incluye.

Por último, el equipo quiere llevar la cuenta de cuántos productos se han creado en total. Esto sugiere la necesidad de una variable a nivel de clase, compartida por todas las instancias.

Para poder llegar con los tiempos de entrega dispuestos por Silvia, será necesario realizar las siguientes acciones:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicios prácticos

1. **Encapsulación:**
   - Convertí los atributos de `Cliente` a `private`.
   - Creá getters y setters para `nombre` y `email`.
   - Asegurá que `email` contenga un `@` antes de asignarlo.
2. **Colaboración entre clases:**
   - Creá una clase `Carrito` que contenga una `ArrayList<Producto>`.
   - Agregale métodos para sumar productos y calcular el total.
   - Demostrá el uso creando un `Carrito`, agregándole productos y mostrando el total.
3. **Variables estáticas:**
   - En `Producto`, agregá una variable estática que lleve la cuenta de cuántos productos se crean.
   - Mostrá ese valor luego de instanciar varios productos.
4. **Control de acceso con setters:**
   - Añadí una validación en el setter de `cantidadEnStock` para que no se permita asignar valores negativos.
   - Probá asignar -10 y verificá que el valor no cambie.

Materiales y recursos adicionales

- Documentación Oficial de Java: Encapsulamiento y Modificadores de Acceso
- Conceptos de OOP: Encapsulación y Colaboración
- Videos recomendados en YouTube sobre encapsulación, getters, setters y composición de objetos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Preguntas para reflexionar

- ¿Cómo el encapsulamiento mejora la robustez del código al evitar modificaciones indebidas de los datos?
- ¿En qué casos conviene usar variables de clase (estáticas) en lugar de atributos de instancia?
- ¿Cómo cambia el diseño de la aplicación cuando pensamos en objetos que colaboran entre sí en vez de datos sueltos?

Próximos pasos

En la siguiente clase, profundizaremos en herencia, polimorfismo y otros conceptos avanzados de POO. Mientras tanto, practicá agregando getters, setters, variables estáticas y relaciones entre clases, viendo cómo mejora la calidad y mantenibilidad de tu software en TechLab. ¡Venís súper!
Hasta la próxima.

Buenos Aires Ciudad