UNIVERSIDAD NEBRIJA

Técnicas de programación avanzada

Curso 2023-2024

Tema 5.2 Inversión de control

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

5.2 Inversión de Control e Inyección de Dependencias

Inversión del control (IoC)

Recordad: *Estructuras de control (Tema 1)*

IoC es un principio de diseño en el que el flujo de control de un programa se invierte en comparación con la programación tradicional. En lugar de que el programador controle el flujo y las dependencias, un framework externo se encarga de ello. Esto se ve comúnmente en frameworks como Spring, donde el contenedor gestiona la creación y la vinculación de objetos.

Imagina un coche que necesita varios componentes para funcionar, como un motor, ruedas, etc. En lugar de que el coche controle cuándo y cómo se crea el motor, un sistema externo (como un taller mecánico, o en nuestro caso, un framework como Spring) toma el control. El coche simplemente espera y utiliza los componentes que se le proporcionan.

La **Inversión de control** puede implementarse mediante:

- *Inyección de dependencias* (se puede crear un contenedor que gestione las instancias u objetos)
- *Eventos* (como en las aplicaciones con GUI)

5.2 Inversión de Control e Inyección de Dependencias

Inyección de dependencias (DI)

Inyección de Dependencias (DI) se refiere a una técnica de diseño de software donde un objeto recibe otras instancias que necesita (sus dependencias) en lugar de crearlas internamente. En Java, la inyección de dependencias se realiza a menudo a través de frameworks como Spring. Es una forma de reducir el acoplamiento entre clases y aumentar la flexibilidad y la capacidad de prueba del código.

Imagina un coche que necesita varios componentes para funcionar, como un motor, ruedas, etc. En lugar de que el coche construya estos componentes internamente, se los proporciona ("inyecta") desde fuera. Esto permite que el coche sea más flexible y fácil de mantener.

5.2 Inversión de Control e Inyección de Dependencias

Inyección de dependencias (DI)

```java
class Motor {
    private String tipo;
    public Motor() {
        this.tipo = "Motor estándar";
    }
    public String getTipo() {
        return tipo;
    }
}
```

```java
class Coche {
    private Motor motor;
    public Coche() {
        this.motor = new Motor();
    }
    public Motor getMotor() {
        return motor;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche();
        System.out.println("Tipo de motor: "
        + coche.getMotor().getTipo());
    }
}
```

GIT: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/e_inyeccion_dependencias_ejemplo_sin

Sin Inyección de Dependencias

En el primer ejemplo, la clase **Coche** crea una instancia de **Motor** directamente dentro de su constructor. Esto hace que Coche esté fuertemente acoplado a la clase Motor.

Con Inyección de Dependencias

En el segundo ejemplo, **Coche** recibe un objeto **Motor** a través de su constructor. Esto reduce el acoplamiento entre **Coche** y **Motor**, y hace que el código sea más flexible y fácil de probar. Además, se utiliza una interfaz **Motor**, lo que permite cambiar la implementación del motor sin modificar la clase **Coche**.

```java
interface Motor {
    String getTipo();
}
```

```java
class MotorEstandar implements Motor {
    private String tipo;
    public MotorEstandar() {
        this.tipo = "Motor estándar";
    }
    @Override
    public String getTipo() {
        return tipo;
    }
}
```

```java
class Coche {
    private Motor motor;
    public Coche(Motor motor) {
        this.motor = motor; // IoD
    }
    public Motor getMotor() {
        return motor;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Motor motor = new MotorEstandar();
        Coche coche = new Coche(motor);
        System.out.println("Tipo de motor: "
        + coche.getMotor().getTipo());
    }
}
```

GIT: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/e_inyeccion_dependencias_ejemplo_con

Ventajas de la Inyección de Dependencias

- **Desacoplamiento**: DI reduce el acoplamiento entre clases. Las clases no crean sus dependencias directamente, sino que las reciben desde el exterior. Esto facilita el cambio y la sustitución de dependencias sin tener que modificar las clases que las utilizan.

- **Facilita las Pruebas**: Al desacoplar las dependencias, DI hace que sea más fácil realizar pruebas unitarias. Puedes inyectar implementaciones de prueba o mock objects en tus clases, lo que permite probar cada clase de manera aislada.

- **Mayor Flexibilidad y Escalabilidad**: DI permite cambiar las implementaciones de las dependencias sin modificar las clases consumidoras. Esto es especialmente útil en proyectos grandes y escalables, donde los cambios son frecuentes.

- **Gestión Centralizada de Dependencias**: Con frameworks de DI como Spring, las dependencias y su configuración se gestionan centralizadamente, lo que facilita su mantenimiento y actualización.

- **Código Más Limpio y Mantenible**: Al separar la creación de objetos de su uso, el código tiende a ser más limpio y fácil de entender. DI fomenta seguir principios de diseño de software sólidos, como el principio de responsabilidad única.

Cuando Usar la Inyección de Dependencias

- **Proyectos de Gran Escala**: En proyectos grandes, donde el manejo de dependencias puede volverse complejo, DI ofrece una gestión eficiente y centralizada de estas.
- **Proyectos que Requieren Alta Mantenibilidad**: Si prevés cambios frecuentes en tu proyecto o necesitas mantenerlo a largo plazo, DI facilita estos procesos.
- **Cuando se Utilizan Frameworks que Favorecen DI**: Si estás trabajando con frameworks modernos como Spring o Jakarta EE, que están diseñados alrededor de la DI, es recomendable aprovechar sus capacidades.
- **Desarrollo de Aplicaciones Web y Empresariales**: En aplicaciones web y empresariales, donde la gestión de diferentes servicios y componentes es compleja, DI ayuda a organizar y desacoplar estas dependencias.

Consideraciones

- DI no es siempre necesaria, especialmente en proyectos pequeños o scripts simples donde puede introducir una complejidad innecesaria.
- Es importante entender bien DI antes de implementarla para evitar sobre complicar el código con inyecciones innecesarias.

Bibliografía

- Deitel, H. M. & Deitel, P. J.(2008). *Java: como programar*. Pearson education. Séptima edición.
- The Java tutorials. https://docs.oracle.com/javase/tutorial/
- Páginas de apoyo para la sintaxis, bibliotecas, etc.:
  - https://www.sololearn.com
  - https://www.w3schools.com/java/default.asp
  - https://docstore.mik.ua/orelly/java-ent/jnut/index.htm

Técnicas de Programación Avanzada

```
exit(); //Gracias!
```