Dependency Injection

Overview

Dependency injection is a design pattern that separates an object's creation and usage by providing its dependencies externally rather than having the object create them internally.

No matter how simple, every application consists of multiple objects that work together to provide a unified user experience. We use the dependency injection principle to get the objects to collaborate and achieve a common objective.

What Is Dependency Injection

- Dependency injection (DI) is a technique objects use to specify their dependencies, such as other objects they interact with. This is achieved by providing constructor arguments, properties, or arguments to a factory method.

- The container then provides the necessary dependencies when creating the object, which is the opposite of traditional object instantiation, where the object determines its dependencies' location or creation, often using the Service Locator pattern. This approach is commonly known as Inversion of Control.

- Dependency injection achieves loose coupling between objects by removing the responsibility of object instantiation and dependency management from the object itself. Instead, the dependencies are provided externally, typically by a DI container or framework, allowing the object to focus solely on its functionality.

- This means that objects can be easily replaced or updated without affecting other system parts, as long as they adhere to the same interface or contract. By reducing the interdependence between objects, DI enables greater flexibility and maintainability of the system, making it easier to test, extend, and evolve.

How Dependency injection works

The following is an overview of how DI works:

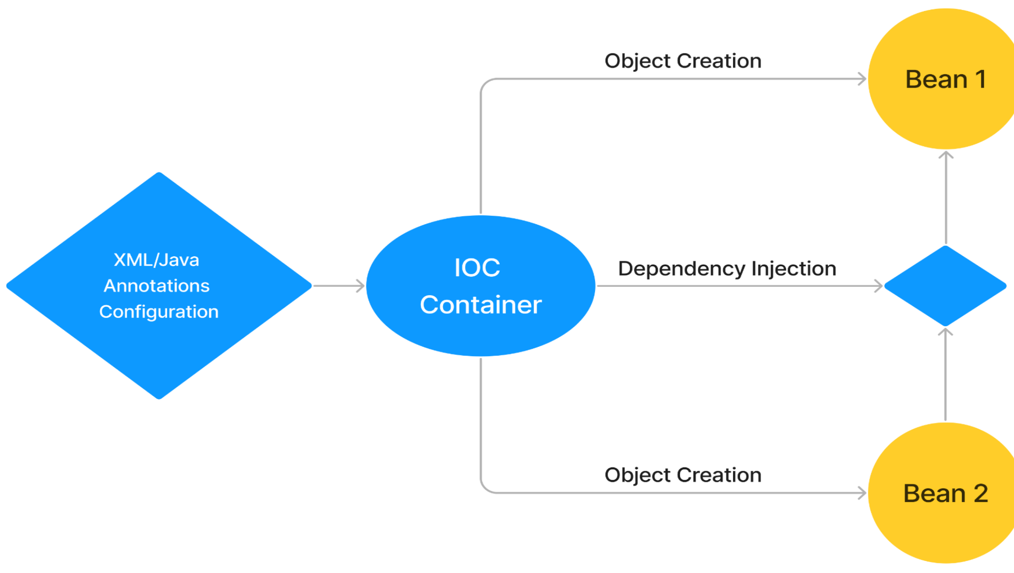
1. **Object Definition:** First, a developer defines the objects and their dependencies using a DI framework or container. The dependencies are constructor arguments, properties, or factory methods.

2. **DI Container Configuration:** The DI container is then configured to manage the dependencies of the objects. This involves registering the objects and their dependencies with the container.

3. **Object Creation:** When an object is requested directly or as a dependency of another object, the container creates it and provides its dependencies.

4. **Dependency Injection:** The container injects the dependencies into the object's constructor or properties, allowing the object to access and use them.

By providing dependencies externally, DI reduces the coupling between objects and improves the modularity and maintainability of the code. It also makes testing easier, as dependencies can be easily mocked or substituted with test-specific implementations.



Types of Dependency Injection

A. Constructor-based Dependency Injection

Constructor-based dependency injection is a dependency injection in which a class's dependencies are provided through its constructor.

In this approach, the class's constructor takes one or more arguments that represent the required dependencies, which are passed in when an instance of the course is created. The constructor then stores these dependencies as instance variables that can be used throughout the class.

Car Dealership Example:

We start by creating a Car interface:

```
public interface Car{
    String showDetails();
    String setDetails();
}
```

We can define two implementations of this interface called FamilyCar and SportsCar

```
public class FamilyCar implements Car{
    private String owner;
    private String brand;
    FamilyCar() {}
    public void FamilyCar(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void setDetails(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void showDetails() {
       return this.owner + “ has a family car of brand ” + this.brand;
    }
}
```

```
public class SportsCar implements Car{
    private String owner;
    private String brand;
    public void SportsCar(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void setDetails(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void showDetails() {
       return this.owner + “ has a Sports car of brand ” + this.brand;
    }
}
```

We Define objects and their dependencies in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">


    <bean id="sports" class="SportsCar">
      <constructor-arg name=”owner” />
       <constructor-arg name=”brand” />
    </bean>

    <bean id="family" class="FamilyCar">
        <constructor-arg name=”owner” />
        <constructor-arg mane=”brand” />
    </bean>
</beans>
```

In the main method, we take the user's name and car choice and show details accordingly.

```
public class Main {
    public static void main(String[] args) {
            System.out.println("Hi welcome please enter your name");
            Scanner scanner = new Scanner(System.in);
            String name = scanner.nextLine();

            System.out.println("Which type of car do you want");
            String type = scanner.nextLine();

            System.out.println("Enter your car Brand");
            String brand = scanner.nextLine();

ClassPathXmlApplicationContext context =
             new ClassPathXmlApplicationContext("applicationContext.xml");
             Car car = (Car) context.getBean(type);
             car.setDetails(name, brand);
             car.showDetails();
    }
}
```

B. Setter-based Dependency Injection

Setter injection is a dependency injection where dependencies are injected into an object using setter methods.

With setter injection, an object is first created using a no-argument constructor, and then its dependencies are set using setter methods. Setter injection can be helpful when we have optional dependencies or when we need to inject multiple dependencies into an object.

In some cases, it can also be easier to read and maintain, as it separates the dependency injection from the object's construction. However, it can be less explicit than constructor injection, making it harder to ensure that all dependencies are correctly set.

In continuation to the above example, we convert it into setter injection

```
public class FamilyCar implements Car{
    private String owner;
    private String brand;
    public void setOwner(String owner){
        this.owner = owner;
    }
    public void setBrand(String brand){
        this.brand = brand;
    }
    public void setDetails(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void showDetails() {
       return this.owner + “ has a family car of brand ” + this.brand;
    }
}
```

```
public class SportsCar implements Car{
    private String owner;
    private String brand;
    public void setOwner(String owner){
        this.owner = owner;
    }
    public void setBrand(String brand){
        this.brand = brand;
    }
    public void setDetails(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }
    public void showDetails() {
       return this.owner + “ has a Sports car of brand ” + this.brand;
    }
}
```

We Define objects and their dependencies in XML using property tag

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">


    <bean id="sports" class="SportsCar">
        <property name=”owner” />
        <property name=”brand” />
    </bean>

    <bean id="family" class="FamilyCar">
        <property name=”owner” />
        <property name=”brand” />
    </bean>
</beans>
```

In the main method, we take the user’s name and car choice and show details accordingly.

```
public class Main {
    public static void main(String[] args) {
            System.out.println("Hi welcome please enter your name");
            Scanner scanner = new Scanner(System.in);
            String name = scanner.nextLine();

            System.out.println("Which type of car do you want");
            String type = scanner.nextLine();

            System.out.println("Enter your car Brand");
            String brand = scanner.nextLine();

ClassPathXmlApplicationContext context =
             new ClassPathXmlApplicationContext("applicationContext.xml");
             Car car = (Car) context.getBean(type);
             car.setDetails(name, brand);
             car.showDetails();
    }
}
```

C. Field Injection

Field injection is a dependency injection technique in Spring Boot that involves injecting dependencies directly into class fields. To use field injection, annotate the field with the `@Autowired` annotation.

When the application starts, Spring Boot will automatically scan for all `@Autowired` fields and inject the appropriate dependencies. Field injection is a convenient way to inject dependencies, but it can make testing more difficult since dependencies cannot be easily swapped out for mocks.

It is generally recommended to use a constructor or setter injection instead, as it they more explicit and testable. However, in some cases, such as with third-party libraries or legacy code, field injection may be necessary.

Example:

```
public class Car{

    @Autowired
    private Engine engine;

}
```

Common Dependency Errors

- **Circular dependencies:** Occur when two or more classes depend on each other. It can result in a deadlock where none of the classes can be instantiated. To avoid this, designing classes with clear responsibilities and avoiding circular dependencies is crucial.

- **Missing dependencies:** This occurs when a required dependency is not available. This can happen when the dependency is not configured correctly or is not present in the application context. It is crucial to ensure that all dependencies are perfectly defined and available.

- **Ambiguous dependencies:** This occurs when multiple dependencies match the required dependency type. This can happen when various interface or abstract class implementations exist. Qualifying dependencies must be qualified using annotations or configured correctly to avoid this.

- **Incorrect dependencies:** Occurs when the wrong dependency is injected into a class. This can happen when the dependency type must be correctly matched to the required type. Ensuring that the correct dependencies are injected in essential classes is essential.

- **Inconsistent dependencies:** This occurs when the dependencies injected into a class are inconsistent with the dependencies used by the class. This can happen when the dependencies need to be appropriately synchronized or when different versions of dependencies are used. To avoid this, it is essential to use a consistent set of dependencies throughout the application.

Instructor Codes

- Car Dealership Application

References:

1. Official spring documentation.
2. Constructor Injection
3. Setter Injection
4. Common Exceptions in Dependency Injection