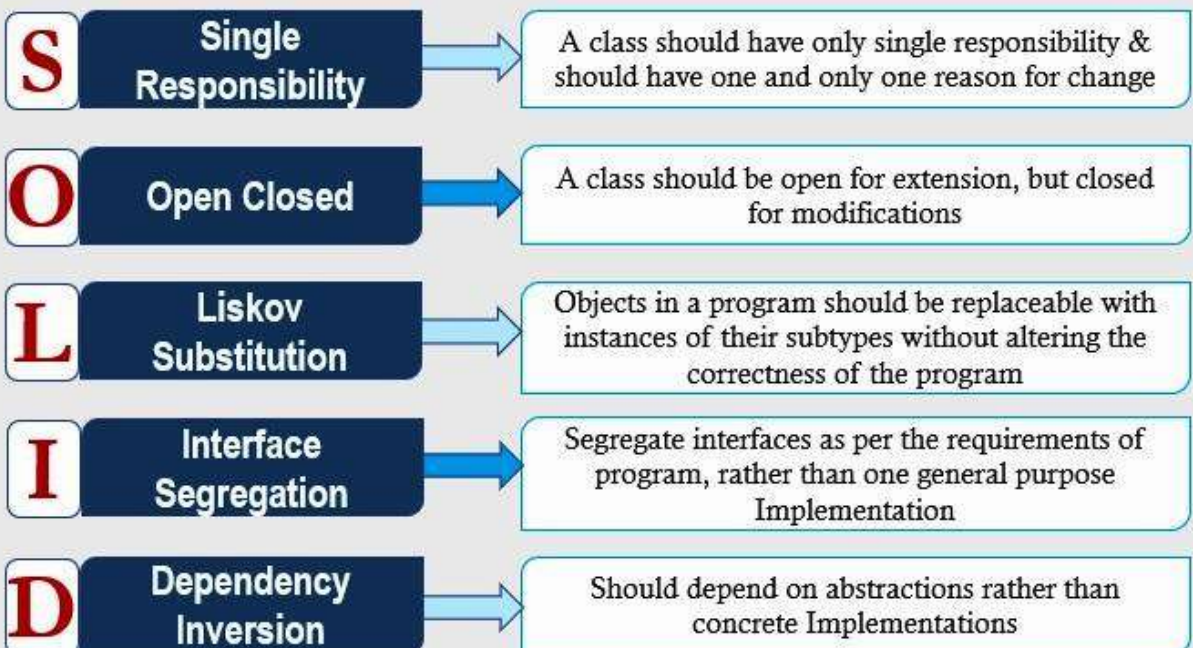


# SOLID PRINCIPLES IN JAVA

## SOLID Principles



## SOLID Design Principles With Examples



## What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern in software development where an object's dependencies (the other objects it needs to function) are injected into it, rather than the object creating its dependencies itself. This is a core concept of Inversion of Control (IoC), and in Spring, it is often achieved through annotations or XML configuration.

## DI in the Context of a Food Delivery App

Without DI (Traditional Approach):

In a food delivery app, you might have different classes like Order, Delivery, Payment, and Restaurant. Without DI, these classes would directly create instances of their dependencies.

For example: **Order.java**

```
package com.hcl.withoutDI;
```

```
public class Order {  
    private Payment payment;  
    private Delivery delivery;  
  
    public Order() {  
        //Traditional Approach  
        // Directly creating dependencies (tight  
coupling)  
        this.payment = new Payment();  
        this.delivery = new Delivery();  
    }  
  
    public void processOrder() {  
        payment.processPayment();  
        delivery.scheduleDelivery();  
    }  
}
```

### Payment.java

```
package com.hcl.withoutDI;

public class Payment {
    public void processPayment() {
        System.out.println("Processing payment...");
    }
}
```

### Delivery.java

```
package com.hcl.withoutDI;

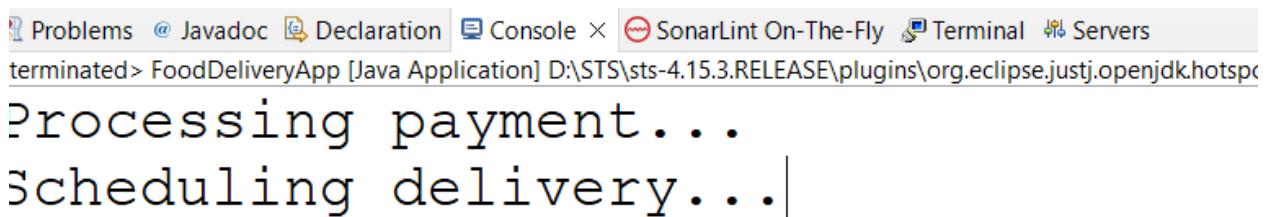
public class Delivery {
    public void scheduleDelivery() {
        System.out.println("Scheduling delivery...");
    }
}
```

### FoodDeliveryApp.java

```
package com.hcl.withoutDI;

public class FoodDeliveryApp {
    public static void main(String[] args) {
        // Create order manually
        Order order = new Order();
        order.processOrder();
    }
}
```

### Output



```
terminated> FoodDeliveryApp [Java Application] D:\STS\sts-4.15.3.RELEASE\plugins\org.eclipse.justj.openjdk.hotspc
Processing payment...
Scheduling delivery...|
```

In the example above, the Order class is tightly coupled to the Payment and Delivery classes. This means if you want to change the payment method (e.g., from Payment to StripePayment), or change the delivery service, you have to modify the Order class, violating the Open/Closed Principle (software entities should be open for extension but closed for

modification).

```
package com.hcl.withoutDI;

public class StripePayment extends Payment {
    @Override
    public void processPayment() {
        System.out.println("Processing payment via
Stripe...");
    }
}
```

## Modify Order.java

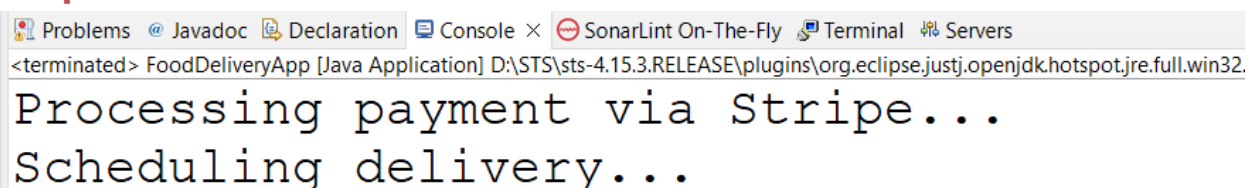
```
package com.hcl.withoutDI;

public class Order {
    private Payment payment;
    private Delivery delivery;

    public Order() {
        //Traditional Approach
        // Directly creating dependencies (tight
coupling)
        this.payment = new Payment();
        this.payment = new StripePayment(); //
Overwrites the previous Payment()
        this.delivery = new Delivery();
    }

    public void processOrder() {
        payment.processPayment();
        delivery.scheduleDelivery();
    }
}
```

## Output

A screenshot of the Eclipse IDE's console window. The window has a title bar with tabs for 'Problems', 'Javadoc', 'Declaration', 'Console', 'SonarLint On-The-Fly', 'Terminal', and 'Servers'. The 'Console' tab is active, showing the output of a Java application. The output consists of two lines: 'Processing payment via Stripe...' and 'Scheduling delivery...'. The console title bar also includes the text '<terminated> FoodDeliveryApp [Java Application] D:\STS\sts-4.15.3.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32'.

```
<terminated> FoodDeliveryApp [Java Application] D:\STS\sts-4.15.3.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32
Processing payment via Stripe...
Scheduling delivery...
```

## Problems with this Approach

1. **Tightly Coupled** - The Order class is dependent on specific implementations of Payment and Delivery. If we change Payment to StripePayment, we must modify the Order class.
2. **Violates Open/Closed Principle** - We should be able to add new implementations without modifying existing code.
3. **Difficult to Unit Test** - We cannot easily mock dependencies during testing.

## Summary

### Problems with this approach:

- **Tightly coupled**: Order knows exactly which Payment and Delivery classes to create.
- **Hard to replace** with other implementations (e.g., StripePayment).
- **Difficult to test** because you can't easily inject mocks.

### With DI (Dependency Injection):

With Dependency Injection, the Order class does not create the Payment or Delivery objects. Instead, these dependencies are injected from the outside (e.g., by the framework or a container). This allows for more flexible and maintainable code.

### Step 1: Define Dependencies

```
package com.hcl.di;

public class Payment {
    public void processPayment() {
        System.out.println("Processing payment...");
    }
}
```

```
}  
  
public class Delivery {  
    public void scheduleDelivery() {  
        System.out.println("Scheduling delivery...");  
    }  
}
```

## Step 2: Create the Order Class with Setter Injection

In this version, the Order class has setter methods that will allow external code (like a Spring container) to inject the dependencies.

// No hard-coded dependency

```
package com.hcl.di;

public class Order {
    private Payment payment;
    private Delivery delivery;

    // Setter methods for dependency injection
    public void setPayment(Payment payment) {
        this.payment = payment;
    }

    public void setDelivery(Delivery delivery) {
        this.delivery = delivery;
    }

    public void processOrder() {
        payment.processPayment();
        delivery.scheduleDelivery();
    }
}
```

Now, Order does **not** create Payment or Delivery. Instead, they will be **injected** by an external system (like Spring).

## Step 3: Configure Dependency Injection (Spring Context)

In Spring, we can configure these dependencies in an XML or Java configuration. For instance, using Spring's ApplicationContext, we would declare beans (objects) and inject dependencies.

```
<?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">

    <!-- Payment beans -->
    <bean id="payment" class="com.hcl.di.Payment"/>
    <bean id="stripePayment" class="com.hcl.di.StripePayment"/>
```

```

<!-- Delivery bean -->
<bean id="delivery" class="com.hcl.di.Delivery"/>

<!-- Order bean using default Payment -->
<bean id="order" class="com.hcl.di.Order">
    <property name="payment" ref="payment"/>
    <property name="delivery" ref="delivery"/>
</bean>

<!-- Another Order bean using StripePayment -->
<bean id="orderWithStripe" class="com.hcl.di.Order">
    <property name="payment" ref="stripePayment"/>
    <property name="delivery" ref="delivery"/>
</bean>
</beans>

```

The **Spring container** will now manage the creation and injection of dependencies.

#### Step 4: Inject and Use Dependencies in the App

Now, when you retrieve the Order bean from the Spring container, Spring will automatically inject the Payment and Delivery dependencies into the Order object.

```
package com.hcl.di;
```

```

public class FoodDeliveryApp {
    public static void main(String[] args) {
        // Load the Spring context
        ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext.xml");

        // Get the Order bean from the context
        Order order = (Order) context.getBean("order");

        // Process the order (with dependencies injected)
        order.processOrder();

        // Get the Order bean with StripePayment
        Order orderWithStripe = (Order) context.getBean("orderWithStripe");

        orderWithStripe.processOrder();
    }
}

```



## Key Benefits of DI in the Food Delivery App:

### 1. Loose Coupling:

- Order, Payment, and Delivery classes are no longer tightly coupled. The Order class no longer needs to know how Payment or Delivery are implemented. If you want to change the payment

system to **StripePayment** or **PayPalPayment**, you can do that without modifying the Order class.

### 2. Flexibility:

- You can easily replace the payment or delivery system (e.g., switch to a different delivery provider) without changing the Order class. You just need to inject a different implementation.

### 3. Easier Testing:

- In unit testing, you can mock the Payment and Delivery dependencies when testing the Order class without having to create real instances of Payment or Delivery.

### 4. Better Maintainability:

- As your app grows and the dependencies between objects increase, DI keeps the code organized and flexible. Changes in one class do not require changes in others.

## Conclusion:

In the context of a food delivery app, Dependency Injection (DI) allows you to manage dependencies between various components (like Order,

Payment, Delivery) in a flexible and decoupled way. By having the Spring container manage these dependencies, you can focus on the business logic, making the app easier to maintain, test, and scale.

