# Dependency_injection

## Overview

Topics Covered:

- Understanding Dependency Injection (DI)
- The Problem of Coupling in Software Design
- Solutions to Reduce Coupling
  - Program to Interfaces
  - Dependency Injection
  - Centralized Configuration

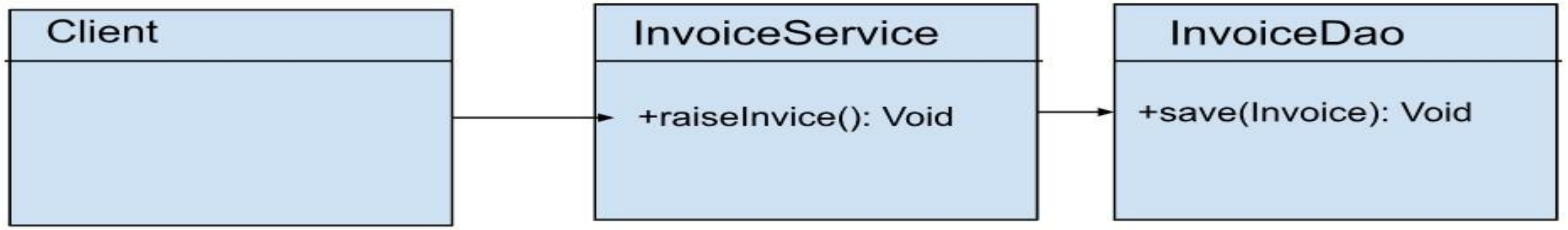
## What is Dependency Injection?

- Dependency Injection (DI) is a core concept in Spring that helps reduce coupling in software.
- It improves maintainability and testability by injecting dependencies instead of creating them manually.
- Designed to solve the problem of tight coupling in applications.

## Understanding Coupling

- Coupling refers to the level of interdependence between components.
- High coupling leads to rigid, difficult-to-maintain applications.
- Example: A service class tightly coupled with a DAO class makes replacing or testing it harder.

# Coupling



Client
InvoiceService
+raiseInvice(): Void
InvoiceDao
+save(Invoice): Void

## Example: Traditional Coupled Code

```
public class InvoiceService {

    private InvoiceDao dao = new InvoiceDao();  // Tight Coupling

}
```

- The above code makes replacing InvoiceDao difficult.
- Changing to Hibernate requires modifying every instance of InvoiceDao.

## Problems with Coupling

- Hard to Change Implementations → Requires modifying multiple classes.
- Difficult to Unit Test → Service classes depend on real database interactions.
- Limited Flexibility → Changing from JDBC to Hibernate requires significant effort.

**Solution:** Reduce Coupling Using Dependency Injection

## Reducing Coupling in Three Steps

1. Program to Interfaces

- Define an interface and implement multiple data access strategies.
- Example:

```
public interface InvoiceDao {

    void save(Invoice invoice);

}
```

## Reducing Coupling in Three Steps

1. Program to Interfaces

- Implementations:

```
public class JdbcInvoiceDao implements InvoiceDao {
    public void save(Invoice invoice) {
        // JDBC Implementation
    }
}

public class HibernateInvoiceDao implements InvoiceDao {
    public void save(Invoice invoice) {
        // Hibernate Implementation
    }
}
```

## Reducing Coupling in Three Steps

1. Program to Interfaces:

```
InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
```

## Reducing Coupling in Three Steps

2. Apply Dependency Injection:

Instead of instantiating dependencies directly, we inject them using setters or constructors.

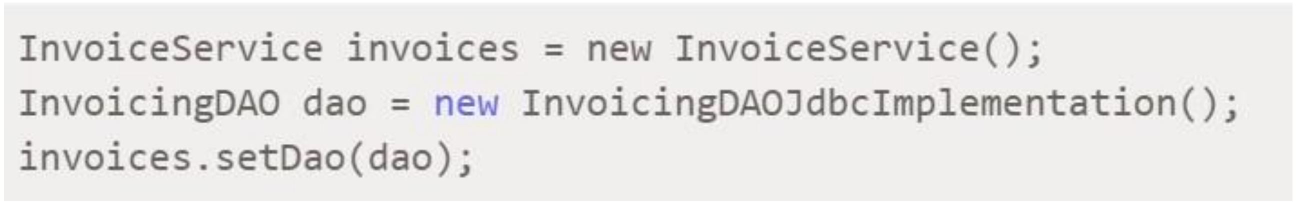
Setter Injection Example:

```
public class InvoiceService {

    private InvoiceDao dao;

    public void setInvoiceDao(InvoiceDao dao) {
        this.dao = dao;
    }
}
```

# Reducing Coupling in Three Steps

2. Apply Dependency Injection:I

In the client class, we can create an instance of the ‘dao’ and instantiate it as a jdbc implementation and tie it to the InvoicingService class:



```
InvoiceService invoices = new InvoiceService();
InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
invoices.setDao(dao);
```

To change the jdbc to hibernate, will be minor change in the client code and only switch the implementation.

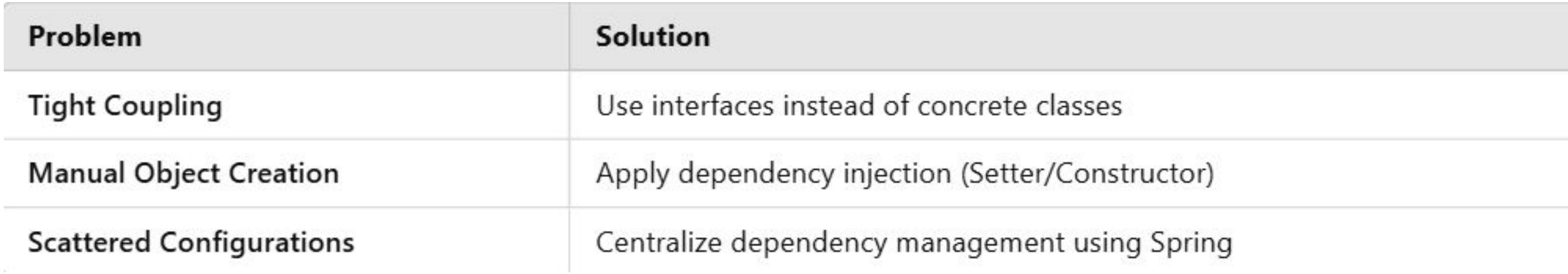
We have now got some flexibility but still we have coupling in our client code. We should do the next step which is centralise configuration and here spring comes in.

## Reducing Coupling in Three Steps

3. Centralized Configuration:

- Dependencies should be managed in one place, not in multiple locations.
- Instead of manually instantiating objects, a container can manage them.
- Container will be called in the client code.
- Client code will remain intact in case of changes in the future.
- Spring automatically injects dependencies from a configuration file or annotations, making applications more flexible and easier to maintain.
- We will look at the Spring container in the next lecture.

## Summary



| Problem | Solution |
|---|---|
| **Tight Coupling** | Use interfaces instead of concrete classes |
| **Manual Object Creation** | Apply dependency injection (Setter/Constructor) |
| **Scattered Configurations** | Centralize dependency management using Spring |