Introduction to Spring Core and Spring Boot

upGrad

Session: Why Spring?

Instructor: Vishwa Mohan

Why Spring?

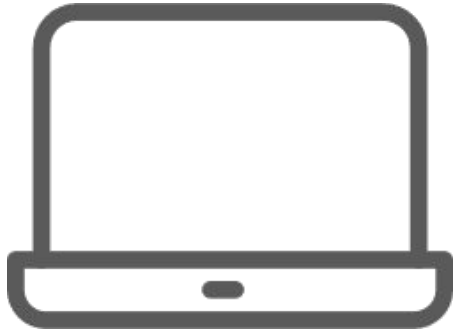
Coupling

COUPLING

Loose Coupling

- It is easy to test and debug if any component malfunctions.
- It is easy to replace or reuse any component with other components.

Tight Coupling



- It is difficult to test and debug if any component stops working.
- It is difficult to replace or use any component with other components.

TIGHT COUPLING

- When multiple classes are **highly dependent** on each other, they are said to be 'tightly coupled'.
- In software development, normally, an object of one class needs an object of another class to execute its functionality. For example, a car needs an engine to execute its functionality.
- However, when you refer a **dependency class** directly inside the dependent class, it introduces **tight coupling**.

```
class Car {
Engine engine;
}
```

```
class Engine {
}
```

Here, the car (**dependent object**) is tightly coupled with the engine (**dependency**).

TIGHT COUPLING

Main Class

```
public class Main {
    public static void main(String[]
    args)
{
     GreetingService greetingService =
       new GreetingService();
       greetingService.greet("John");
     }
}
```

GreetingService Class

```
public class GreetingService {
    public void greet(String name) {
     System.out.println("Hello, " +
         name);
     }
}
```

Dependent Class

Dependency

TIGHT COUPLING

- In the previous example, observe that the **Main** class was directly referring to the **GreetingService** class. Hence, the **Main class** was **tightly coupled** with the **GreetingService class.**
- In the future, if you want to change the greeting service to some other service, for example, one that greets in French, you would need to make changes in the Main class as well.

LOOSE COUPLING

- You can make loosely coupled applications by introducing the following:
  - Interfaces
  - Factory pattern

INTERFACE

```
public interface GreetingService {
    public void greet(String name);
}
```

```
public class EnglishGreetingService
implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, " +
name);
    }
}
```

```
public class FrenchGreetingService
implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Bonjour, " +
name);
    }
}
```

FACTORY PATTERNS

```
public class GreetingServiceFactory {
   public GreetingService getGreetingService(String language) {
       if (language.equals("english")) {
           return new EnglishGreetingService();
       } else if (language.equals("french")) {
           return new FrenchGreetingService();
       } else {
           throw new RuntimeException("No greeting Service exist for " + language + "
           language.");
       }
   }
}
```

FACTORY PATTERNS

```
public class Main {

    public static void main(String[] args) {
        GreetingServiceFactory greetingServiceFactory= new GreetingServiceFactory();
        GreetingService greetingService =
greetingServiceFactory.getGreetingService("french");
        greetingService.greet("John");
    }
}
```

LOOSE COUPLING

- Now, the **Main** class is **NOT** tightly coupled with its dependencies, that is, the **EnglishGreetingService** class. or the **FrenchGreetingService** class.
- You can **make changes** in the **dependency classes** without making any change in the Main class.
- You can also **create more** greeting services and **use them** in your Main class through GreetingServiceFactory **without significantly changing the code** in the Main class.

Inversion of Control (IoC)

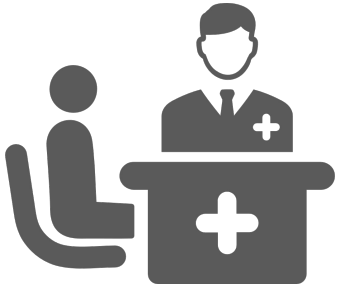
INVERSION OF CONTROL

Normal Flow

Doctor to Patient

- Do this, do that and then go here, etc.
- **The patient is in control** because the doctor is only instructing the course of treatment, and the patient needs to execute it.

Inverted Flow

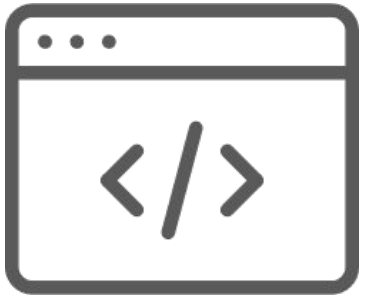


Patient to Doctor

- Just do whatever you want to do with me and help me get rid of this disease!
- **The doctor is in control** because the patient has asked him to take care of the entire course of treatment.

INVERSION OF CONTROL

Normal Flow



- In a classical code, the person who is writing the software instructs the computer 'Do this, do that and then, go here, etc.'
- **The coder is in control!**

Inverted Flow

- The coder who is writing the software fills out a template for the framework, and the framework decides what it will do.
- **The framework is in control!**

NORMAL FLOW

- The normal flow of execution is as follows:
  - The **developer creates a class** named ‘EnglishGreetingService’.
  - The **developer creates an instance of the class** ‘EnglishGreetingService’ in the Main class.
  - The **developer invokes a method defined** in the ‘EnglishGreetingService’ class for execution in the Main class.

NORMAL FLOW

Main Class

```
public class Main {
    public static void main(String[]
    args)
{
    GreetingService  greetingService
= new  GreetingService();
greetingService.greet("John"  );
    }
}
```

EnglishGreetingService Class

```
public class EnglishGreetingService
implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, "
    + name);
    }
}
```

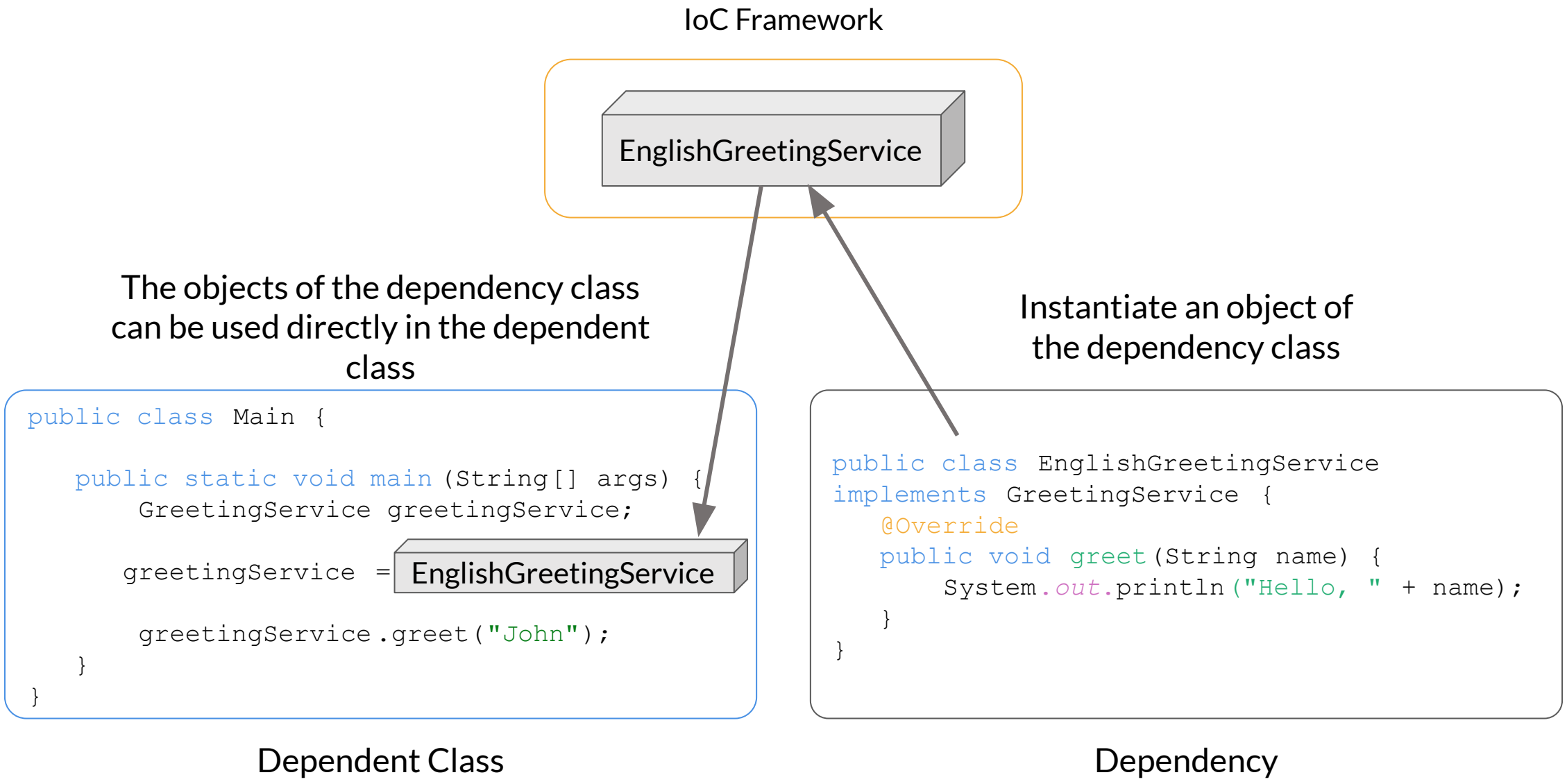
Dependent Class

Dependency

INVERSION OF CONTROL

- Flow of execution when using the IoC Framework:
  - The **developer creates a class** named ‘EnglishGreetingService’.
  - The **IoC Framework creates an instance** of ‘EnglishGreetingService’ that can be used inside the Main class.
  - The **developer invokes a method defined** in the ‘EnglishGreetingService’ class for execution in the Main class.

INVERSION OF CONTROL (IoC)



Dependency Injection (DI)

DEPENDENCY INJECTION (DI)

- Dependency Injection (DI) helps you to inject dependencies into an object.
- IoC is a generic programming principle, whereas DI is an implementation of IoC.
- IoC states that the dependent classes should not create objects of the dependency classes. Dependency objects would be created by either the IoC container or the Framework and provided to the dependent object.
- IoC does not describe how dependency objects will be provided to the dependent objects.

DEPENDENCY INJECTION (DI)

- DI is one of the implementation or a subtype of IoC and instructs how to provide dependency objects to the dependent objects.
- DI states that dependencies would be injected into the dependent objects via the setter method or a constructor.

Introduction to Spring

WHAT IS SPRING?

- Previously you learnt how to make applications **loosely coupled** by writing extra codes in the form of the **factory pattern** or by **creating interfaces.**
- For big enterprise applications, this **boilerplate code** can get quite **complex** and divert you from the main goal of the application.
- **Spring Framework** helps you by **providing** this boilerplate code.

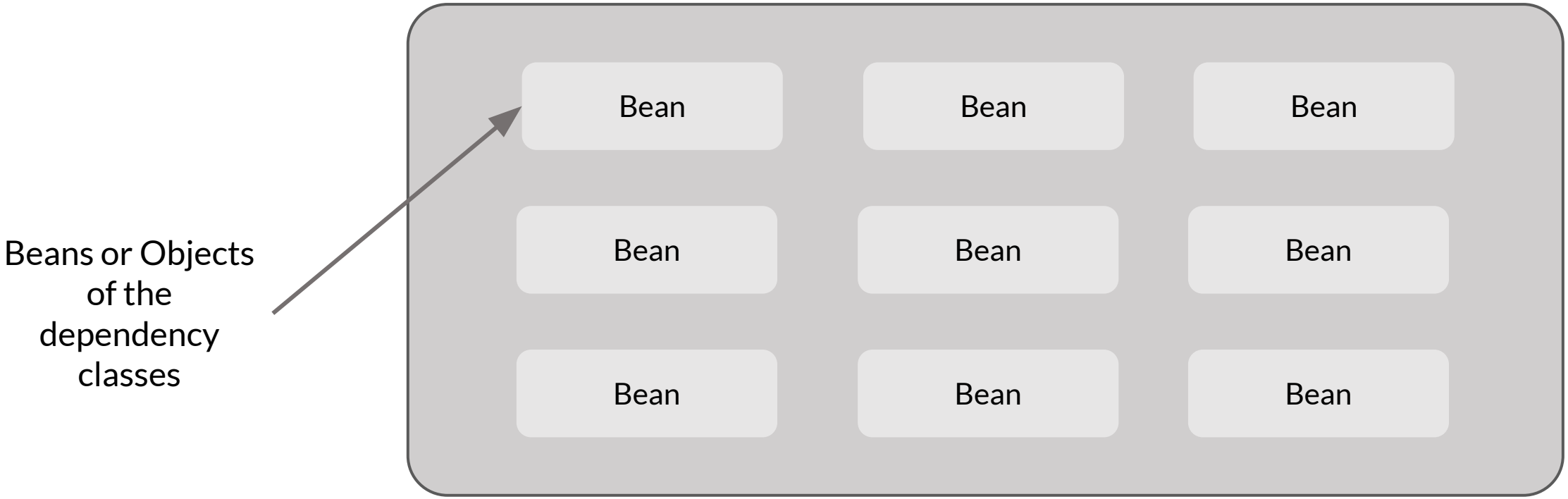
ADVANTAGES OF USING A FRAMEWORK

- A **framework**, or a **software framework**, provides you with a well-tested generic boilerplate code written by some of the best software developers by following design principles and best practices.
- This will not only **save time** but also insert the **correct components** in our applications. This is **fast** and **easy** approach.
- Almost all frameworks, including **Spring**, rely on the concept of **Inversion of Control (IoC).**

SPRING CONTAINER

- The IoC Framework **instantiates objects** of the dependency classes that can be used by the dependent classes.
- The IoC Frameworks behave like containers for such objects, so IoC Frameworks are also called **IoC containers.**
- The **IoC container** of the **Spring Framework** is called, for obvious reasons, called the Spring container.
- In Spring, the **dependency objects** contained inside the Spring container are called **Spring beans** or, simply, **beans.**

SPRING CONTAINER



Spring Container

SPRING CONTAINER

- Spring provides the following two distinct types of containers:
  - Spring BeanFactory Container
  - Spring ApplicationContext Container

BEAN FACTORY

- It is similar to a factory class that contains a collection of methods to get beans from the Spring container.
- It is lightweight, as it instantiates beans only when requested by the application, instead of creating all the beans at the start of the application.
- Being lightweight, it is generally used for mobile device applications.

APPLICATIONCONTEXT

- It is an interface that extends the BeanFactory interface.
- ApplicationContext is a heavyweight interface compared to BeanFactory, as it loads all the beans at the startup of the application.
- It provides some extra enterprise-specific functionality on top of the BeanFactory container.
- Since it includes all the functionalities of the BeanFactory container and also provides some additional facilities, it is generally recommended over BeanFactory wherein the startup time is not an issue, such as when writing the server-side code.

FEATURES OF SPRING FRAMEWORK

- It is **open source.**
- It is **lightweight** in terms of execution, as it does not require an external container support to run applications, such as the Servlet container that is used to run the Servlet/JSP application, and because of its POJO implementation.
- **Modular:** It is developed in different modules. So, you can import only those required Spring modules that are needed as per the requirements of the application and ignore the rest.
- The integration with other frameworks is **loosely coupled,** such as integration with the Hibernate Framework.

FEATURES OF SPRING FRAMEWORK

- **Aspect-oriented programming** (AOP): It uses AOP to separate cross-cutting concerns (logging, security, exception handling, etc.) from the business logic.
- **Dependency injection:** It helps in developing loosely coupled applications and, in turn, facilitates easier unit testing.
- **IoC containers:** It creates and manages the lifecycle of the Spring beans.
- **Transaction management:** It is used to maintain transactions and is mostly used to maintain database transactions.

SPRING ARCHITECTURE