Java Spring Framework

Introduction to the Spring Framework

- The course begins with an introduction to the **Spring Framework**, which is a core technology for building enterprise-level Java applications.
- Although **Spring Boot** is closely associated with Spring, it will be introduced later.
- At this stage, the focus is only on understanding **what Spring Framework is**, before discussing Spring Boot.
- **Spring Framework and Spring Boot are different**, but Spring Boot is built on top of the Spring Framework.

Purpose of Using Frameworks in Application Development

- Frameworks are commonly used when building **large or enterprise-level applications**, regardless of the programming language.
- In Java, **Spring** is one of the most widely used and powerful frameworks for application development.
- Before Spring, developers used multiple separate frameworks:
  - **EJBs (Enterprise JavaBeans)** for enterprise applications
  - **Struts** for building web applications
  - **Hibernate** for ORM (Object Relational Mapping) and database interactions
- Spring unified these capabilities into a **single framework**, simplifying development.

Key Characteristics of the Spring Framework

- Spring is described as a **lightweight framework**.
  - The concept of "lightweight" will be explained later in the course.
- It primarily works with **POJOs (Plain Old Java Objects)**.
  - Simple Java objects can perform complex tasks when managed by Spring.
- Spring is designed specifically for **enterprise-level application development**.

Spring Framework Overview from Spring.io

- The official Spring website (**spring.io**) highlights several core goals:
  - Spring makes Java **productive**

    - Spring supports **reactive programming**
    - Spring makes Java **simple and modern**
- Spring enables developers to build applications using modern programming approaches.

Spring as an Ecosystem

- Spring is **not a single module or tool**.
- Initially, Spring focused mainly on **Dependency Injection**.
- Over time, Spring expanded into multiple projects, forming a complete **ecosystem**.
- Spring provides support for building:
    - **Microservices**
    - **Reactive applications**
    - **Cloud-based applications**
    - **Web applications**
    - **Serverless applications**
    - **Event-driven systems**
    - **Batch processing applications**

Spring Code Simplicity and Spring Boot

- Spring-related examples often show **simple annotation-based code** such as a “Hello World” application.
- These examples usually use **Spring Boot**, which simplifies Spring configuration.
- Spring Boot is not the same as Spring Framework but **makes Spring easier to use**.
- The course will later cover:
    - Core Spring syntax
    - How Spring works without Spring Boot
    - How Spring Boot simplifies development

Spring Projects and Modules

- Spring consists of many well-known projects available under the **Spring Projects** section:
    - **Spring Framework**
    - **Spring Boot**
    - **Spring Data**
    - **Spring Cloud**
    - **Spring Security**
    - **Spring for GraphQL**
    - **Spring Batch**

- The Spring ecosystem continues to grow, with new projects being added over time.
- Spring also supports multiple platforms and languages:
  - Android application development
  - Scala integration
  - Kotlin support as a programming language for Spring

Course Roadmap and Learning Path

- This course will cover:
  - **Spring Framework**
  - **Spring Boot**
  - **Spring AOP**
  - **Spring Security**
  - Microservices using Spring
- The learning journey is structured to gradually build understanding.
- As the course progresses, it becomes clear why **Spring is widely used for enterprise applications** and why it is considered a powerful and practical framework.

Why Spring Is Popular Despite Other Framework Options

- There are **multiple frameworks available** in the industry besides Spring.
- A framework becomes widely adopted based on three key factors:
  - **Features**: The framework must provide useful and powerful features that help developers build applications efficiently.
  - **Community**: A strong and active community helps with support, learning, and long-term adoption.
  - **Documentation**: Clear, detailed, and well-maintained documentation is essential for developers.

Spring's Strengths Compared to Other Frameworks

- **Spring Framework** is considered a great framework because it satisfies all three critical factors:
  - It offers **rich features** that support enterprise application development.
  - It has a **large and active community**, making it widely used across the industry.
  - It provides **excellent documentation**, which helps developers learn

and troubleshoot effectively.

Importance of Spring Documentation

- Apart from learning through this course, developers are encouraged to **refer to the official Spring documentation**.
- The official documentation is described as **very well written and easy to understand**, especially for those who prefer reading.

Accessing Spring Official Documentation

- The Spring documentation can be accessed by searching for **"Spring Docs"** on Google.
- The first search result typically leads to the **official Spring documentation page**.
- At the time of creating the videos:
  - The Spring Framework version in use is **6.1.1**
  - Other versions, such as **6.1.2**, are also available for reference.
- The version to follow may vary depending on when the content is being viewed.

Using Documentation for Specific Topics

- The documentation is organized by **topics and modules**.
- For example:
  - If you want to learn about **Spring AOP**, you can directly navigate to the AOP section.
  - Each topic is explained clearly with proper structure.
- This makes the documentation a reliable learning resource alongside video-based learning.

Additional Learning Resources

- Besides the official documentation:
  - There are **books available in the market** related to Spring.
- However, the **official Spring documentation alone is sufficient and highly recommended** as a primary reference.

Prerequisites Before Learning the Spring Framework

- Spring is a **framework**, so certain foundational knowledge is required before starting.

- Without these prerequisites, understanding Spring concepts can become difficult.

Core Java Knowledge Requirements

- Strong understanding of **Core Java** is essential, including:
  - Java **syntax**
  - **OOP concepts** (Object-Oriented Programming)
  - **Exception handling**
  - **Threads**
    - Advanced threading is not mandatory
    - Basic understanding will be sufficient
  - **Collections API**
- These concepts are used extensively throughout the Spring Framework.

Importance of JDBC (Java Database Connectivity)

- Enterprise applications heavily rely on **data**.
- To work with data, applications must interact with **databases**.
- In Java, database communication is handled using **JDBC**.
- JDBC acts as a bridge between:
  - Java applications
  - Relational databases
- Basic knowledge of JDBC is required to understand data-related operations in Spring.

Build Tool Knowledge (Maven / Gradle)

- Every Spring project requires a **build tool**.
- Common build tools include:
  - **Maven**
  - **Gradle**
- In this course:
  - **Maven** will be used
- Understanding Maven concepts is necessary for managing dependencies and building projects.

ORM and Hibernate Basics

- The course will cover **Spring ORM** concepts.
- For this:
  - Basic understanding of **ORM (Object Relational Mapping)** is required
  - Knowledge of **Hibernate** or any ORM framework is helpful
- ORM helps in mapping Java objects to database tables.

Servlets and Web Application Basics

- To build web applications using Spring, **Servlet knowledge** is useful.
- Although **Servlets are considered outdated** for direct application development:
  - Modern applications use **Spring MVC** instead
- However:
  - Spring MVC runs on servers like **Tomcat**
  - **Tomcat is a servlet container**
- Therefore, understanding servlets helps in:
  - Knowing how Spring MVC works internally
  - Understanding request–response handling

Summary of Required Prerequisites

- Core Java fundamentals
- JDBC
- Maven
- ORM / Hibernate basics
- Servlets (basic understanding)

Conclusion

- If you are familiar with these prerequisites:
  - You are **ready to start learning Spring Framework**
- These concepts form the foundation on which Spring is built.

Software Requirements for Developing a Spring Application

- To develop and run a **Spring application**, certain software tools are required.
- The primary requirements include:
  - **JDK (Java Development Kit)**
  - An **IDE (Integrated Development Environment)**

———

Java Development Kit (JDK) Requirement

- Since Spring is a Java framework, **JDK must be installed** on the system.
- JDK is required to:

        - Compile Java code
        - Run Java applications
    - Installation steps for JDK are covered separately in the Java section.
    - To verify JDK installation:
        - Open **Command Prompt / Terminal**
        - Run: java -version

    - For **Spring Framework 6**, the requirement is:
        - **Java 17 or above**
    - Newer versions such as **Java 21** also work without issues.

Choice of IDE (Integrated Development Environment)

- Multiple IDEs are available for Java and Spring development:
    - **Eclipse**
    - **IntelliJ IDEA**
    - **VS Code**
    - **NetBeans**
- No single IDE is mandatory.
- The **code, configuration, and project structure remain the same** regardless of the IDE used.
- This consistency is because:
    - The project uses **Maven**
    - Maven enforces a standard project structure

Use of Maven for Project Structure

- Maven creates a **standard directory structure** for Spring projects.
- Because of Maven:
    - IDE choice does not affect project layout
    - Code and configuration remain identical across IDEs

Using Eclipse for Spring Development

Downloading Eclipse

- Eclipse can be downloaded from the official Eclipse website.
- Available packages include:
    - **Eclipse IDE for Java Developers**
        - Suitable for core Java development
    - **Eclipse IDE for Enterprise Java and Web Developers**

        - Recommended for web and enterprise applications
    - Other Eclipse packages are not required for this course.

Spring Support in Eclipse

- By default, Eclipse does **not include Spring support**.
- To add Spring support:
    - Open **Help → Eclipse Marketplace**
    - Search for **Spring Tool**
    - Install **Spring Tools (Spring Tool 4)**
- This installs Spring-related plugins.
- After installation:
    - Eclipse provides options to create **Spring Boot projects**
- Installation requires:
    - Default settings
    - Restart of Eclipse

Using VS Code for Spring Development

- **VS Code** also supports Java and Spring development through extensions.
- Required extensions:
    - **Spring Tools (by VMware)**
    - Optional: **Spring Boot Dashboard**
- Installing the VMware Spring Tools extension often installs required dependencies automatically.
- After installing extensions:
    - Spring projects can be created directly from VS Code
- VS Code is supported but **not used in this course**

Spring Tools and IDE Integration

- Earlier, Spring provided a standalone IDE called **Spring Tool Suite (STS)**.
- STS was based on Eclipse and focused only on Spring development.
- Now, Spring focuses on:
    - **Plugins and extensions** instead of standalone IDEs
- Spring tools are available as extensions for:
    - Eclipse
    - VS Code
    - Other supported IDEs

Using IntelliJ IDEA for Spring Development

IntelliJ Versions

- IntelliJ IDEA has two versions:
  - **Community Edition (Free)**
  - **Ultimate Edition (Paid)**
- Most companies provide licenses for the **Ultimate Edition**.
- The **Community Edition**:
  - Does **not provide built-in Spring support**
  - Does **not offer Spring plugins**
- The **Ultimate Edition**:
  - Provides direct **Spring and Spring Boot support**

Course Usage Decision

- IntelliJ IDEA will be used in this course.
- Spring projects will be created using:
  - **Maven archetypes**
- Even without direct Spring support:
  - Project setup and code will work correctly
- IDE switching is common in real-world development:
  - Developers often switch IDEs, languages, and tools as needed

———

Summary of IDE Options

- **Eclipse**
  - Free
  - Requires Spring Tools plugin
  - Provides Spring Boot project creation support
- **IntelliJ IDEA**
  - Community: Free, no Spring support
  - Ultimate: Paid, full Spring support
- **VS Code**
  - Lightweight
  - Requires Spring extensions
- Any IDE can be used as long as:
  - JDK is installed
  - Maven is used

Core Concepts Before Learning Spring: IoC and Dependency Injection

- Before starting Spring concepts, it is essential to understand **two foundational concepts**:
  - **IoC (Inversion of Control)**
  - **DI (Dependency Injection)**
- These are **different concepts**, but they are **closely related** and work together in Spring.

Inversion of Control (IoC)

- **IoC** stands for **Inversion of Control**.
- It means **transferring control from the programmer to another entity**.
- Traditionally, as a programmer, you are responsible for:
  - Creating objects
  - Controlling the application flow
  - Managing object lifecycle (creation, usage, destruction)

Traditional Object Creation Problem

- In core Java, objects are created using the new keyword.
- Example:

```
Laptop laptop = new Laptop();
```

- While object creation is simple:
  - The programmer must create the object
  - Maintain it
  - Destroy it when no longer needed
- This puts **full control** in the hands of the programmer.

Why Object Creation Control Is a Problem

- Managing objects takes focus away from the **business logic**.
- The **business logic** is what differentiates one application from another.
- The goal of a developer should be:
  - Focus on business logic
  - Not worry about object creation and management

What Is Inversion of Control in Practice

- With IoC:
  - The programmer **gives up control** of object creation
  - Another entity takes over this responsibility
- This means:
  - Control is **inverted**
  - Object creation and lifecycle are handled externally
- IoC applies not only to object creation but also to **application flow control**

IoC Container in Spring

- Spring implements IoC using an **IoC Container**.
- The IoC container can be imagined as a **box** that:
  - Holds all application objects
  - Manages their lifecycle
- Responsibilities of the Spring IoC Container:
  - Creating objects
  - Storing objects
  - Managing dependencies
- The programmer does **not** manually create objects anymore.

Dependency Injection (DI)

- **Dependency Injection** is the mechanism used to implement IoC.
- DI is a **design pattern**, not a principle.
- While IoC defines *what* needs to be achieved, DI defines *how* it is achieved.

Understanding Dependency with an Example

- Consider two classes:
  - Laptop
  - CPU
- A Laptop **depends on** a CPU.
- Without a CPU, a laptop cannot function.
- Inside the Laptop class, a CPU object is required.

How Dependency Injection Works Conceptually

- Both Laptop and CPU objects are created by Spring.
- Both objects exist inside the **IoC container**.
- The challenge:
  - How to connect the CPU object to the Laptop object?
- This connection is achieved through **Dependency Injection**.
- Spring injects the required dependency automatically.

Relationship Between IoC and Dependency Injection

- **IoC**:
  - A **principle**
  - Defines the idea of giving control to the framework
- **Dependency Injection**:
  - A **design pattern**
  - Used to implement the IoC principle
- In practice:
  - These terms are often used interchangeably
  - Conceptually, they serve different purposes

Spring Project and Dependency Injection

- Spring consists of multiple projects.
- One of the core Spring projects focuses on:
  - **Dependency Injection**
- This project is responsible for:
  - Managing objects
  - Injecting dependencies
  - Implementing IoC

Conclusion

- **IoC** removes object creation responsibility from the developer.
- **Dependency Injection** connects dependent objects automatically.
- Together:
  - They allow developers to focus on **business logic**
  - They form the foundation of the Spring Framework
- The actual implementation of these concepts in Spring will be covered in

upcoming sections.

Creating the First Spring Boot Application

- There are two possible ways to begin learning Spring:
  - Start with **Spring Framework** and later move to **Spring Boot**
  - Start with **Spring Boot** to see how easy it is, then understand what happens behind the scenes
- This course follows the **second approach**:
  - Start with **Spring Boot**
  - Later explore internal Spring Framework concepts
- In this section:
  - A **basic Spring Boot application** is created
  - Dependency Injection will be demonstrated in later videos

---

Choosing the IDE for the First Spring Boot Project

- Multiple IDEs are available, but for this step:
  - Start with **Eclipse**
  - Later move to **IntelliJ IDEA**
- Eclipse is chosen first because:
  - It supports Spring Boot directly using Spring Tools

Creating a Spring Boot Project Using Eclipse

- Open **Eclipse**
- Click on **Create a Project**
- Instead of creating a plain Maven project:
  - Choose **Spring Starter Project**
- This option becomes available because **Spring Tools** is installed

---

Spring Initializer Configuration in Eclipse

- Eclipse internally uses the **Spring Initializer** service:
  - URL: **start.spring.io**
- Project configuration:

        - **Project Name**: spring-boot-first
        - **Build Tool**: Maven
        - **Packaging**: Jar (default)
        - **Java Version**: 17
        - **Language**: Java
        - **Group ID**: com.telusko
        - **Artifact ID**: spring-boot-first
        - **Package Name**: com.telusko.app
    - Click **Next** to proceed

Selecting Spring Boot Version and Dependencies

- Spring Boot version:
    - Select **3.2**
    - Older versions (e.g., 1.6) are available but not used
- Dependency selection:
    - Spring allows **modular dependency selection**
    - Examples of available modules:
        - Spring Web
        - WebSocket
        - JPA
    - For this first project:
        - **No additional dependencies are selected**
- Click **Finish** to generate the project

Generated Spring Boot Project Structure

- The project is downloaded from **start.spring.io**
- A complete Maven-based Spring Boot project is created
- The pom.xml file contains:
    - **Spring Boot Starter dependency**
- This single dependency is sufficient to:
    - Run a basic Spring Boot application
    - Use Dependency Injection features

Limitation of IntelliJ IDEA Community Edition

- IntelliJ IDEA Community Edition:
    - Does **not** provide built-in Spring project creation
    - Spring support is available only in **Ultimate Edition**
- Because of this:

- Spring Boot projects cannot be directly created inside IntelliJ Community

---

Creating a Spring Boot Project Using start.spring.io

- To use IntelliJ Community Edition:
  - Go directly to **start.spring.io**
- Configure the project:
  - **Project**: Maven
  - **Language**: Java
  - **Spring Boot Version**: 3.2
  - **Group ID**: com.telusko
  - **Artifact ID**: spring-boot-demo
  - **Packaging**: Jar
  - **Java Version**: 17
  - **Dependencies**: None
- Click **Generate**
- A **ZIP file** is downloaded

Opening the Project in IntelliJ IDEA

- Extract the downloaded ZIP file
- Open **IntelliJ IDEA**
- Click **Open**
- Select the extracted project folder
- IntelliJ loads the Maven-based Spring Boot project successfully

---

Understanding the Spring Boot Project Dependencies

- The project contains many dependencies under **External Libraries**
- pom.xml shows:
  - **spring-boot-starter**
- Spring Boot internally includes:
  - **Spring Framework**
  - Version shown is **Spring Framework 6**
- This confirms:
  - **Spring Boot 3 runs on Spring Framework 6**

Running the First Spring Boot Application

- The project already contains a main class:
    - Annotated Spring Boot application class
- No explanation of annotations yet
- Add a simple output statement:
    - Print "Hello World"
- Run the application

Application Startup Output

- Spring Boot displays:
    - ASCII-style Spring banner
    - Spring Boot version
    - Java version (Java 17)
- The application runs successfully
- "Hello World" output confirms:
    - The application is running correctly

Implementing Dependency Injection: Spring vs Spring Boot

- Before writing actual code for **Dependency Injection**, it is important to understand the relationship between:
    - **Spring Framework**
    - **Spring Boot**
- These are **not separate technologies**.
- **Spring Boot is built on top of the Spring Framework**.

Why Spring Boot Was Introduced

- When **Spring Framework** was first introduced:
    - It was powerful and widely adopted
    - It benefited both developers and enterprises
- However, even for a **simple Hello World application**, Spring required:
    - Manual project creation
    - Extensive configuration
    - XML configuration files
    - Explicit bean definitions
- This made:

    - Initial setup time-consuming
    - Entry-level learning more difficult

Configuration Complexity in Core Spring

- In traditional Spring:
    - You cannot run a simple application with minimal code
    - You must:
        - Configure XML files
        - Define beans
        - Set up application context
- These steps:
    - Will be explained later in the course
    - Are powerful but require effort and time

Spring Boot as a Solution

- **Spring Boot** was introduced to simplify Spring development.
- It is an **opinionated framework**, meaning:
    - It follows predefined conventions
    - It makes assumptions to reduce configuration
- Spring Boot provides:
    - A ready-to-run project structure
    - Minimal or no configuration
    - Faster application startup
- A Spring Boot project:
    - Works on the first run
    - Requires very little setup

Spring vs Spring Boot Clarification

- **Spring Framework**
    - Core framework
    - Provides dependency injection and core features
- **Spring Boot**
    - Built on top of Spring Framework
    - Simplifies configuration and setup
- Even when using Spring Boot:
    - You are still using **Spring Framework internally**

Version Alignment

- In this course:
  - **Spring Framework version**: 6
  - **Spring Boot version**: 3
- Important relationship:
  - **Spring Boot 3 runs on Spring Framework 6**

Industry Usage Perspective

- In modern application development:
  - Most projects are built using **Spring Boot**
- However:
  - Understanding **Spring Framework internals** is essential
  - It helps in understanding what Spring Boot does behind the scenes

---

Learning Approach in This Course

- First:
  - Dependency Injection will be implemented using **Spring Boot**
- Then:
  - The same concepts will be explained using **Spring Framework**
- This approach provides:
  - Ease of learning with Spring Boot
  - Deep understanding of core Spring concepts

Conclusion

- Spring Boot simplifies Spring application development
- Spring Framework remains the foundation
- Dependency Injection is implemented using Spring, even when Spring Boot is used
- This course will cover:
  - Both **ease of use** (Spring Boot)
  - And **internal working** (Spring Framework)

Implementing Dependency Injection Using Spring and Spring Boot

---

Introduction: Spring vs Spring Boot

- **Spring Framework** is the core framework that provides features like **Dependency Injection (DI)** and **Inversion of Control (IoC)**.
- **Spring Boot** is built **on top of Spring Framework** to reduce configuration and speed up project setup.
- Spring Boot is **opinionated**, meaning it provides sensible defaults so applications can run with minimal setup.
- Even when using Spring Boot, you are still fundamentally using the **Spring Framework**.
- In this setup:
  - **Spring Framework version:** Spring 6
  - **Spring Boot version:** Spring Boot 3 (built on Spring 6)

Why Spring Boot Was Introduced

- Early Spring applications required:
  - Manual project setup
  - XML configuration files
  - Explicit bean definitions
- Even a simple **Hello World** required extensive configuration.
- Spring Boot simplifies this by:
  - Auto-configuring the project
  - Providing a ready-to-run application structure
  - Reducing boilerplate configuration

Goal: Understanding Dependency Injection

- The objective is to:
  - First implement **Dependency Injection (DI)** using **Spring Boot**
  - Then understand what happens **behind the scenes** using the **Spring Framework**
- This approach helps understand both:
  - Ease of use with Spring Boot
  - Core concepts of Spring Framework

Environment Update

- Previously, the application was running on **Java 17**

- The setup has been updated to **Java 21**
- All examples going forward will use **Java 21**

Spring Boot Application Entry Point

- The application starts with a **main class** annotated with:
  - **@SpringBootApplication**
- Inside the main method:
  - SpringApplication.run(...) is called
- This method:
  - Boots the Spring application
  - Starts the **Spring container**
  - Returns an object of **ConfigurableApplicationContext**

Understanding the Spring Container

- Spring uses an **IoC (Inversion of Control) container**
- The container:
  - Creates objects
  - Manages their lifecycle
  - Injects dependencies
- Objects created and managed by Spring are called **Beans**
- A **Bean** is just a normal Java object with a special name in Spring terminology

---

Accessing the IoC Container

- SpringApplication.run() returns an **ApplicationContext**
- ApplicationContext allows communication with the IoC container
- Example:
  - Store the return value of run() in an ApplicationContext variable
  - Use it to request beans from the container

Creating a Simple Class (Alien)

- A simple class named **Alien** is created
- It contains:
  - A non-static method that prints a message
- Initially:
  - The object is created using new Alien()

- This works, but it is **not Dependency Injection**

Problem with Manual Object Creation

- Using new Alien():
  - The object is created by the developer
  - Spring has no control over it
- Goal of DI:
  - Let **Spring create and manage objects**
  - Avoid using new keyword

Requesting an Object from Spring

- Use:
  - context.getBean(Alien.class)
- This asks the Spring container:
  - "Give me the bean of type Alien"
- At this point:
  - Spring throws an error:
    - **NoSuchBeanDefinitionException**

Why the Bean Was Not Found

- By default:
  - Spring does **not** create objects for all classes
- Reason:
  - Applications may contain hundreds of classes
  - Spring should only manage selected ones

Making a Class a Spring Bean

- Use the annotation:
  - **@Component**
- Adding @Component on a class:
  - Tells Spring to:
    - Create the object
    - Manage it
    - Store it in the IoC container

Effect of @Component Annotation

- Once @Component is added:
  - Spring automatically detects the class
  - Creates the bean at startup
- Now:
  - context.getBean(Alien.class) works successfully
  - The method call executes as expected

How Dependency Injection Is Happening

- Spring:
  - Creates the Alien object
  - Stores it in the IoC container
- The application:
  - Requests the object using ApplicationContext
- This process is the foundation of **Dependency Injection**

Creating Multiple Bean Requests

- Calling context.getBean(Alien.class) multiple times:
  - Works without errors
  - The method executes each time
- A question is raised:
  - Are these the **same object** or **different objects**?
- This topic is intentionally deferred for later discussion

Key Observations

- Only classes annotated with **@Component** become Spring-managed beans
- Classes without this annotation:
  - Will not be available in the IoC container
- Dependency Injection works because:
  - Spring controls object creation
  - The application only requests objects

Code Summary

Main Application Class

- Starts Spring Boot
- Retrieves ApplicationContext
- Requests Alien bean from IoC container
- Calls methods on the injected bean

Alien Class

- Annotated with **@Component**
- Contains a method to confirm successful bean creation
- Managed entirely by Spring

```
package com.springdemostarter.firstproject;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class FirstprojectApplication {

    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(FirstprojectApplication.class, args);
        System.out.println("Hi Sai,Welcome to Java Spring
Framework");


        // Alien alien = new Alien(); // YOU control creation
-- ❌ Without Spring:
        // As we know that everytime whenever the spring
creates the objects for us, it will be available in the
container which is IOC
        // We have to find the way to get to the container, we
have to do that is by using application context.
        // SpringApplication.run(FirstprojectApplication.class,
args);
        // If we click on run it returns the object of
ConfigurableApplicationContext which is extending the
(ApplicationContext, Lifecycle, Closeable)
        // As it means if we simply use the ApplicationContext
```

```
return value we get the ApplicationContext object which is
context
        // what we did is run returns the context obj --
ApplicationContext context =
SpringApplication.run(FirstprojectApplication.class, args);
        // Now we got the  way to communicate with the IOC
container. We can simply say hey context that means hey
container, give me the object
        // and remember the object is not object. In spring the
object is called bean

        Alien obj1 = context.getBean(Alien.class); // // SPRING
controls creation -- ✅ With Spring:
        // Here we need to mention which class object that we
want. As we do have multiple classes we need to specify the
class name saying that
        // I need the object of the alien class
        obj1.testingBeans();

        // Creating the object multiple times
        Alien obj2 = new Alien();
        obj2.testingBeans();
        // Actually it will work. But the question is are we
getting the same object or are we getting different object?
        System.out.println(obj1 == obj2);
        // ✅ Answer: SAME object (by default) -- true
        // Spring beans are singleton by default.
        // (We'll change this later using @Scope("prototype")
    }

}
```

```
package com.springdemostarter.firstproject;

import org.springframework.stereotype.Component;

@Component
// Now making this class as a component, you are making sure
that your spring knows that spring has to manage this
particular object.
// So create the object, assemble the object and manage it.
Everything will be done by spring, just by this annotation.
public class Alien {
```

```
    public void testingBeans()
    {
        System.out.println("Object got successfully created
with the help of spring -- Application context");
    }
}
```

Adding a Dependency Layer: Wiring Beans with @Autowired in Spring Boot

---

Extending the Dependency Injection Concept

- Previously:
  - The **main method** accessed the **Spring container** using ApplicationContext
  - An **Alien** object was retrieved using getBean()
- Now:
  - Introduce a new dependency where **Alien depends on another object**
  - This demonstrates **multi-layer dependency injection**

Real-World Analogy

- A **programmer (Alien)** needs a **machine (Laptop)** to write and compile code
- This means:
  - Alien cannot function independently
  - Alien requires a Laptop object to perform its task

Creating the Laptop Class

- A new class **Laptop** is introduced
- It contains:
  - A simple method to simulate compiling or coding
- Example behavior:
  - Prints "Started coding..."

Problem: Laptop Dependency Is null

- The Alien class declares a Laptop reference
- When calling alien.code():
  - A **NullPointerException** occurs
  - Error indicates the Laptop object is null
- Reason:
  - Spring created the Laptop bean
  - But did **not inject it** into Alien

Annotating Laptop as a Spring Bean

- Add **@Component** to the Laptop class
- Effect:
  - Spring now creates and manages the Laptop object
- Verification:
  - Retrieving Laptop directly using context.getBean(Laptop.class) works
  - Confirms the bean exists in the IoC container

Why Injection Still Fails Inside Alien

- Even though Laptop exists in the container:
  - Spring does not automatically connect it to Alien
- Spring requires **explicit instructions** to wire dependencies

---

Introducing Wiring with @Autowired

- **Wiring** means connecting dependent objects together
- Use **@Autowired** on the dependency field in Alien
- Purpose:
  - Instruct Spring to search the container
  - Inject the matching Laptop bean automatically

Alien Class with Dependency Injection

- Alien is annotated with **@Component**
- Laptop field is annotated with **@Autowired**
- Spring responsibilities:
  - Create the Alien object

- Find the Laptop bean
- Inject it into Alien

Successful Dependency Resolution

- After adding @Autowired:
  - alien.code() executes successfully
  - Output confirms "Started coding..."
- This proves:
  - Spring handled object creation
  - Spring handled dependency wiring

Dependency Chain Overview

- **Main Method**
  - Depends on Alien
- **Alien**
  - Depends on Laptop
- **Laptop**
  - Performs the actual work
- Each dependency is resolved by Spring automatically

Key Rule to Remember

- Outside the main method:
  - You **do not** have direct access to the container
- Therefore:
  - You must use **@Component** to register beans
  - You must use **@Autowired** to inject dependencies

Scalability of This Approach

- You can add more layers:
  - Example: Laptop depends on CPU
- Steps:
  - Create a CPU class
  - Annotate it with @Component
  - Inject it into Laptop using @Autowired
- This creates a clean dependency chain managed by Spring

Configuration Styles Mentioned

- Spring applications can be configured using:
  - **XML-based configuration**
  - **Java-based configuration**
  - **Annotation-based configuration**
- The current approach uses:
  - **Annotation-based configuration**
  - Especially streamlined in **Spring Boot**

Takeaway

- Spring Boot simplifies Dependency Injection by:
  - Auto-detecting components
  - Auto-wiring dependencies
- However:
  - Understanding **Spring Framework internals** is crucial
- The next step:
  - Dive deeper into **Spring Framework**
  - Understand what happens **behind the scenes** in the IoC container

```
package com.springdemostarter.firstproject;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class FirstprojectApplication {

    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(FirstprojectApplication.class, args);
        System.out.println("Hi Sai,Welcome to Java Spring
Framework");

        // Alien alien = new Alien(); // YOU control creation
-- ❌ Without Spring:
```

```
        // As we know that everytime whenever the spring
creates the objects for us, it will be available in the
container which is IOC
        // We have to find the way to get to the container, we
have to do that is by using application context.
        // SpringApplication.run(FirstprojectApplication.class,
args);
        // If we click on run it returns the object of
ConfigurableApplicationContext which is extending the
(ApplicationContext, Lifecycle, Closeable)
        // As it means if we simply use the ApplicationContext
return value we get the ApplicationContext object which is
context
        // what we did is run returns the context obj --
ApplicationContext context =
SpringApplication.run(FirstprojectApplication.class, args);
        // Now we got the  way to communicate with the IOC
container. We can simply say hey context that means hey
container, give me the object
        // and remember the object is not object. In spring the
object is called bean

        Alien obj1 = context.getBean(Alien.class); // // SPRING
controls creation -- ✅ With Spring:
        // Here we need to mention which class object that we
want. As we do have multiple classes we need to specify the
class name saying that
        // I need the object of the alien class
        obj1.testingBeans();

        // Creating the object multiple times
        Alien obj2 = context.getBean(Alien.class);
        obj2.testingBeans();
        // Actually it will work. But the question is are we
getting the same object or are we getting different object?
        System.out.println(obj1 == obj2);
        // ✅ Answer: SAME object (by default) -- true
        // Spring beans are singleton by default.
        // (We'll change this later using @Scope("prototype")


        // Firstly we need to annotate our Laptop class with
@Component to create the laptop object inside the container
        // If we do obj2.code which throws Cannot invoke
```

```
"com.springdemostarter.firstproject.Laptop.code()" because
"this.laptop" is null
        // Even though Laptop obj was created it is throwing
null. To prove that Laptop object got created
        // Laptop lap= context.getBean(Laptop.class);
        // lap.code(); // which successfully got printed -
Started coding...
        // That means the object got created, and we are able
to use that in the main because we do have the container access
        // Then the question arrives why it is not working with
obj.code(). The reason is it is not AutoWired
        obj2.code();


        // The flow of what happening here is we can have
multiple layers, and we got alien, and then we got laptop.
        // So basically our main is dependent on the alien
object. Alien is dependent on the laptop object.
        // But when you're using alien in the main you can use
the context directly is because we have the access.
        // But apart from the main, whenever you want to use
this object creation or object accessing, you have to use
@Component and @Autowired
    }

}
```

```
package com.springdemostarter.firstproject;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
// Now making this class as a component, you are making sure
that your spring knows that spring has to manage this
particular object.
// So create the object, assemble the object and manage it.
Everything will be done by spring, just by this annotation.
public class Alien {

    @Autowired
    // If we mentioned Autowired so now your spring framework
knows that it is their responsibility to search for this laptop
```

```
object inside the container
    Laptop laptop;

    public void testingBeans()
    {
        System.out.println("Object got successfully created
with the help of spring -- Application context");
    }

    public void code()
    {
        laptop.code();
    }
}


package com.springdemostarter.firstproject;

import org.springframework.stereotype.Component;

@Component
public class Laptop {

    public void code() {
        System.out.println("Started coding...");
    }
}
```