1. What is spring?

Spring is a **lightweight Java framework** used to build enterprise applications.
Its core feature is **Dependency Injection (DI)**, which helps achieve **loose coupling**.

2. Spring Core Module

The Core module provides:

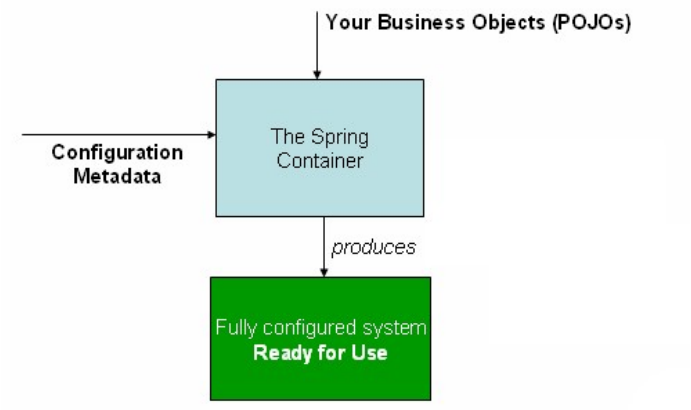
- IoC Container
- Dependency Injection
- Bean lifecycle management

Key packages:

- org.springframework.beans
- org.springframework.context

3. Inversion of Control (IoC)

IoC means the **Spring container creates and manages objects**, not the programmer.



Spring Container

Responsible for managing beans.

Types of Containers:

1. BeanFactory

- Basic container
- Lazy initialization
- Lightweight

2. ApplicationContext

- Advanced container
- Eager initialization
- Supports:
    - Internationalization
    - Event handling
    - AOP integration

Common implementations:

- ClassPathXmlApplicationContext
- AnnotationConfigApplicationContext

Spring Bean

A Spring Bean is an object managed by the Spring container.

Bean Lifecycle

1. Spring creates the bean
2. Dependencies are injected
3. Initialization callbacks
4. Bean is ready to use
5. Destruction (when container shuts down)

Spring Container = Object factory + Dependency manager + Lifecycle controller

Without IoC (Tightly Coupled)

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Car {
    Engine engine = new Engine();  // tightly coupled
}
```

With IoC (Spring manages objects)

Spring creates the Engine object and injects it into Car.

4. Dependency Injection (DI)

DI is the process of **providing dependencies to a class from outside**.

Setter Injection (Example)

```
class Car {
    private Engine engine;

    public void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

XML Configuration

```
<bean id="engine" class="Engine"/>
<bean id="car" class="Car">
    <property name="engine" ref="engine"/>
</bean>
```

Constructor Injection (Example)

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    void drive() {
        engine.start();
        System.out.println("Car is moving");
    }
}
```

XML Configuration

```
<bean id="engine" class="Engine"/>
<bean id="car" class="Car">
    <constructor-arg ref="engine"/>
</bean>
```

Configuration Types

1. XML Configuration
2. Annotation-Based Configuration
3. Java-Based configuration

1. XML Configuration

```
public class Engine {
    private String type;
    private int horsepower;

    // setters
    public void setType(String type) {
        this.type = type;
    }

    public void setHorsepower(int horsepower) {
        this.horsepower = horsepower;
    }

    @Override
    public String toString() {
        return "Engine [type=" + type + ", horsepower=" + horsepower + "]";
    }
}
```

applicationContext.xml

```
<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="engine" class="com.example.Engine">
        <property name="type" value="V8"/>
        <property name="horsepower" value="450"/>
    </bean>

</beans>
```

2. Annotation-Based Configuration Example

Using stereotype annotations like @Component, @Controller ,@Service and @Repository

Java class:

```
@Component
public class Engine {

    @Value("Electric")
    private String type;

    @Value("300")
    private int horsepower;

    @Override
    public String toString() {
        return "Engine [type=" + type + ", horsepower=" + horsepower + "]";
    }
}
```

Enable component Scanning in XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Enable component scanning -->
    <context:component-scan base-package="com.example"/>

</beans>
```

Enable Component Scanning using java:

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

3. Java-Based Configuration Example

Java class:

```
public class Engine {
    private String type;
    private int horsepower;

    public void setType(String type) {
        this.type = type;
    }

    public void setHorsepower(int horsepower) {
        this.horsepower = horsepower;
    }

    @Override
    public String toString() {
        return "Engine [type=" + type + ", horsepower=" + horsepower + "]";
    }
}
```

Java Configuration Class

```
@Configuration
public class AppConfig {

    @Bean
    public Engine engine() {
        Engine engine = new Engine();
        engine.setType("Hybrid");
        engine.setHorsepower(350);
        return engine;
    }
}
```

What is Autowiring in Spring?

Autowiring is how Spring automatically injects one bean into another.

Ways to Autowire in Spring

Spring supports **4 main ways**:

1. **By Type** (most common)
2. **By Name**
3. **Constructor Injection (recommended)**
4. **Qualifier-based**

1. Field Injection (@Autowired on field)

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public void save() {
        userRepository.save();
    }
}
```

How Spring resolves it

- Looks for a bean of type UserRepository
- Injects it automatically

2. Constructor Injection (BEST PRACTICE)

```
@Service
public class UserService {

    private final UserRepository userRepository;

    @Autowired   // optional if only one constructor
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. Setter Injection

```
@Service
public class UserService {

    private UserRepository userRepository;

    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

4. Autowiring by Name (@Qualifier)

Problem: Multiple beans of same type

```
@Component
public class EmailService { }

@Component
public class SmsService { }
```

Both implement:

```
public interface NotificationService {
    void notifyUser();
}
```

Injection error

```
@Autowired
private NotificationService notificationService;
```

Spring says: NoUniqueBeanDefinitionException

Solution: @Qualifier

```
@Autowired
@Qualifier("emailService")
private NotificationService notificationService;
```

Autowiring Without @Autowired (Spring Boot)

Spring Boot automatically autowires constructors only

```
@Service
public class OrderService {

    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```
