SPRING

Spring Core

Definition
Spring Core is the **foundation module of the Spring Framework** that provides **IoC (Inversion of Control)** and **Dependency Injection (DI)** to build **loosely coupled Java applications**.

Key Points
- Manages **object creation and lifecycle**
- Reduces **tight coupling** between classes
- Uses **IoC Container** (BeanFactory / ApplicationContext)
- Supports **XML, Annotation, and Java-based configuration**

Purpose / Importance
- Makes applications **easy to maintain and test**
- Improves **code reusability**
- Allows **easy change of implementation** without changing logic
- Widely used in **enterprise & microservice applications**

Core Concepts in Spring Core
- **IoC (Inversion of Control)**
- **Dependency Injection**
- **Beans**
- **Spring Container**
- **Bean Lifecycle**
- **Configuration**

Inversion of Control (IoC)
Definition
IoC means **Spring controls object creation**, not the developer.
Key Points
- Object creation is shifted from programmer to Spring
- Spring decides **when and how objects are created**
- Achieved using **Spring Container**

Real-life Example
You order food in a restaurant 🍽️.
You don’t cook it → the **kitchen (Spring)** prepares it and gives it to you.

Dependency Injection (DI)
Definition
Dependency Injection is a technique where **Spring injects required objects into a class automatically**.
Types of DI
- **Constructor Injection**
- **Setter Injection**
- **Field Injection (using @Autowired)**

Purpose
- Removes hard-coded dependencies
- Improves testing and flexibility

Real-life Example
A car needs an engine
Instead of creating the engine itself, the **engine is provided from outside.**

Spring Bean
Definition
A Spring Bean is an **object managed by the Spring IoC container.**
Key Points

- Created, initialized, and destroyed by Spring
- Defined using:
  - @Component
  - @Service
  - @Repository
  - @Controller

Spring Container
Definition
Spring Container is responsible for **managing beans and dependencies.**
Types

- **BeanFactory** – Lightweight, basic
- **ApplicationContext** – Advanced (most used)

Why ApplicationContext?

- Supports AOP, events, internationalization
- Better for enterprise applications

Bean Lifecycle (Important for Interview)
Steps

1. Bean Instantiation
2. Dependency Injection
3. Initialization
4. Bean Ready for Use
5. Destruction

Configuration Types
1. XML Configuration

- Old style
- Verbose

2. Annotation Configuration (Most Used)

- @Component
- @Autowired
- @Configuration
- @Bean

3. Java-based Configuration

- Pure Java, no XML
- Clean and readable

Common Interview Questions
Q1: What is Spring Core?
Spring Core provides IoC and Dependency Injection to create loosely coupled Java applications.
Q2: Difference between IoC and DI?

- **IoC** → Concept (control given to Spring)
- **DI** → Implementation of IoC

Q3: What is a Bean?
A Bean is an object managed by the Spring container.

Q4: Why Spring is preferred?
Because it reduces coupling, improves testability, and supports enterprise applications.

One-Line Summary (For Freshers)
Spring Core helps developers write clean, flexible, and maintainable Java applications by managing objects and their dependencies automatically.

Spring Modules

Definition
Spring Framework is divided into **multiple modules**, where each module provides **specific functionality**, and developers can use **only what they need**.

Key Points

- Spring is **modular**, not monolithic
- Each module solves a **specific problem**
- Modules are **independent but integrated**
- Reduces application size and complexity

Purpose / Importance

- Easy to learn and use step-by-step
- Improves performance by loading only required modules
- Makes Spring suitable for **small to large enterprise applications**

Main Spring Modules (High-Level)

1. **Core Container**
2. **AOP**
3. **Data Access / Integration**
4. **Web**
5. **Messaging**
6. **Test**

---

1. Core Container Module

Includes

- spring-core
- spring-beans
- spring-context
- spring-expression (SpEL)

Purpose

Provides **IoC, DI, Bean management, and configuration support**

Real-life Example

Like the **foundation of a building** – everything stands on it.

---

2. AOP (Aspect-Oriented Programming)

Definition

AOP is used to **separate cross-cutting concerns** like logging and security from business logic.

Key Uses

- Logging
- Transaction management
- Security

- Exception handling

Real-life Example

CCTV in a mall – works everywhere without being part of each shop.

---

3. Data Access / Integration Module

Includes

- JDBC
- ORM (Hibernate, JPA)
- Transactions
- JMS

Purpose

Simplifies database interaction and **reduces boilerplate code**

Real-life Example

ATM machine handling transactions safely with database support.

---

4. Web Module

Includes

- Spring MVC
- REST support
- WebSocket

Purpose

Used to build **web applications and REST APIs**

Real-life Example

Website frontend talking to backend

---

5. Messaging Module

Purpose

Enables **asynchronous communication** between applications.

Examples

- JMS
- Message Queues

Real-life Example

WhatsApp messages – sender and receiver don't need to be online at the same time.

---

6. Test Module

Purpose

Supports **unit and integration testing** of Spring applications.

Supports

- JUnit
- Mockito
- TestContext Framework

Real-life Example

Mock exams before real interview

Spring Module Dependency Flow (Easy to Remember)

Core → AOP → Data → Web → Test

Common Interview Questions

Q1: Why is Spring divided into modules?

To provide flexibility and allow developers to use only required features.

Q2: Which is the most important Spring module?

Core Container module.

Q3: Which module is used for REST APIs?

Spring Web (Spring MVC).

Q4: Which module handles database operations?
Data Access / Integration module.

One-Line Summary
Spring modules allow developers to build powerful applications by choosing only the features they need.

Spring Major Modules

Core | AOP | Data | MVC | Security

1. Spring Core

Definition
Spring Core is the **foundation module** that provides **IoC and Dependency Injection.**
Key Points

- Manages objects (Beans)
- Reduces tight coupling
- Uses IoC Container
- Supports annotations & configuration

Purpose
To create **loosely coupled, maintainable Java applications**
Real-life Example
Electricity board supplies power instead of each house generating its own.

---

2. Spring AOP

Definition
Spring AOP handles **cross-cutting concerns** separately from business logic.
Key Points

- Uses Aspects, Advice, Join Points
- Common for logging, security, transactions
- Improves code cleanliness

Purpose
To **avoid code duplication** across methods
Real-life Example
Office attendance system applied to all employees automatically.

---

3. Spring Data

Definition
Spring Data simplifies **database access and persistence logic.**
Key Points

- Supports JDBC, JPA, Hibernate
- Reduces boilerplate code
- Uses repositories

Purpose
To make database operations **easy and consistent**
Real-life Example
Bank passbook automatically records transactions.

---

4. Spring MVC

Definition
Spring MVC is a **web framework** based on the **Model–View–Controller pattern.**
Key Points

- Handles HTTP requests

- Separates UI, logic, and data
- Supports REST APIs

Purpose
To build **web applications and RESTful services**
Real-life Example
Restaurant system
Waiter (Controller) → Kitchen (Model) → Customer (View)

---

5. Spring Security
Definition
Spring Security provides **authentication and authorization.**
Key Points
- Secures URLs & APIs
- Supports roles & permissions
- Protects against CSRF, XSS

Purpose
To prevent **unauthorized access**
Real-life Example
Office ID card system – only authorized people can enter.

Quick Comparison Table (Interview Favorite)

Module	Main Use
Core	Object management
AOP	Cross-cutting concerns
Data	Database operations
MVC	Web & REST
Security	Authentication & Authorization

Common Interview Questions
Q1: Which module is mandatory in Spring?
Spring Core.
Q2: Which module is used for logging?
Spring AOP.
Q3: Which module interacts with DB?
Spring Data.
Q4: Which module handles HTTP requests?
Spring MVC.
Q5: Which module secures APIs?
Spring Security.

One-Line Memory Trick
Core builds → AOP manages → Data stores → MVC serves → Security protects

---

Spring Container

Definition:
Spring Container is the **core component of Spring** that is responsible for **creating, managing, and destroying Spring Beans.**

Key Responsibilities

- Creates objects (Beans)
- Injects dependencies
- Manages bean lifecycle
- Handles configuration

Purpose / Importance

- Removes manual object creation (new keyword)
- Ensures **loose coupling**
- Improves **testability and maintainability**

Types of Spring Containers

1. BeanFactory

Definition

A lightweight container that provides **basic IoC and DI support.**

Key Points

- Lazy initialization (creates bean when requested)
- Used in simple applications
- Less features

---

2. ApplicationContext

Definition

An advanced container used in **enterprise applications.**

Key Points

- Eager initialization
- Supports AOP, events, i18n
- Most commonly used

BeanFactory vs ApplicationContext

Feature	BeanFactory	ApplicationContext
Initialization	Lazy	Eager
Enterprise support	No	Yes
AOP & Events	No	Yes
Usage	Rare	Most common

How Spring Container Works (Flow)

1. Read configuration (XML / Annotations / Java Config)
2. Create beans
3. Inject dependencies
4. Manage lifecycle
5. Destroy beans on shutdown

Real-life Example

Apartment maintenance office
Residents don’t manage repairs → office handles everything.

Configuration Ways Used by Container

- XML-based
- Annotation-based (@Component, @Autowired)
- Java-based (@Configuration, @Bean)

Common Interview Questions

Q1: What is Spring Container?

It manages the lifecycle and dependencies of Spring beans.

Q2: Which container is mostly used?
ApplicationContext.
Q3: Why ApplicationContext is better?
It supports enterprise features like AOP and event handling.
Q4: Does Spring Container create all beans?
Yes, based on configuration.

One-Line Summary
Spring Container is the heart of Spring that controls object creation and dependency management.

Spring Containers
ApplicationContext & BeanFactory

BeanFactory

Definition
BeanFactory is the **basic Spring container** that provides **IoC and Dependency Injection.**

Key Points

- Lightweight container
- Uses **lazy initialization**
- Creates bean **only when requested**
- Fewer enterprise features

Purpose
Used for **simple or memory-constrained applications**

Real-life Example
Small kirana shop– items are brought only when a customer asks.

ApplicationContext

Definition
ApplicationContext is an **advanced Spring container** built on top of BeanFactory.

Key Points

- Uses **eager initialization**
- Supports **AOP, events, i18n**
- Better exception handling
- Most widely used container

Purpose
Designed for **enterprise-level applications**

Real-life Example
Big supermarket– all items are stocked before opening.

Initialization Difference (Very Important)

Container	Initialization
BeanFactory	Lazy
ApplicationContext	Eager

Feature Comparison (Interview Favorite)

Feature	BeanFactory	ApplicationContext
IoC & DI	Yes	Yes
Lazy loading	Yes	No

Feature	BeanFactory	ApplicationContext
AOP support	No	Yes
Event handling	No	Yes
i18n	No	Yes
Enterprise usage	No	Yes

When to Use What

Use BeanFactory when

- Application is very small
- Memory usage must be minimal

Use ApplicationContext when

- Building web or enterprise apps
- Need AOP, security, transactions

Common Interview Questions

Q1: Which container is recommended in Spring?

ApplicationContext.

Q2: Why BeanFactory is rarely used?

It lacks enterprise features.

Q3: Does ApplicationContext extend BeanFactory?

Yes.

One-Line Memory Trick

BeanFactory = Basic, ApplicationContext = Advanced

Inversion of Control (IoC)

Definition

Inversion of Control is a principle where the **control of object creation and dependency management is given to Spring**, instead of the programmer.

Key Points

- Removes usage of new keyword
- Spring creates and manages objects
- Improves loose coupling
- Implemented using **Spring Container**

Purpose / Importance

- Makes code **flexible and maintainable**
- Easy to replace implementations
- Simplifies testing (mocking)

How IoC Works (Flow)

1. Developer defines configuration
2. Spring Container reads configuration
3. Container creates objects
4. Dependencies are injected
5. Application uses ready objects

Relationship Between IoC and DI

Term	Meaning
IoC	Concept / Principle
DI	Technique to implement IoC

Real-life Example
Mobile charging
You don’t generate electricity → power company supplies it.

Without IoC vs With IoC
Without IoC
- Class creates its own dependencies
- Tight coupling
- Hard to test

With IoC
- Dependencies provided by Spring
- Loose coupling
- Easy to test

Common Interview Questions
Q1: What is IoC?
Giving control of object creation to the Spring framework.
Q2: How is IoC achieved in Spring?
Using Dependency Injection.
Q3: What manages IoC in Spring?
Spring Container.

One-Line Summary
IoC means **“Don’t create objects, let Spring create them.”**

Constructor Injection

Definition
Constructor Injection is a type of **Dependency Injection** where **dependencies are provided through the class constructor.**

Key Points
- Dependencies are injected at **object creation time**
- Ensures required dependencies are **not null**
- Recommended approach by Spring
- Supports **immutability**

Purpose / Importance
- Makes classes **more reliable**
- Easier to test (constructor arguments can be mocked)
- Prevents incomplete object creation

How It Works
1. Spring creates the object
2. Calls the constructor
3. Injects required dependencies

4. Object is ready to use

Simple Code Example

```
@Component
public class Car {

    private Engine engine;

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

Spring automatically injects Engine into Car.

Real-life Example

Laptop charger

You must plug in the charger **while starting** the laptop.

Constructor Injection vs Setter Injection

Feature	Constructor Injection	Setter Injection
Injection time	Object creation	After creation
Mandatory dependency	Yes	No
Immutability	Yes	No
Recommended	Yes	No

Common Interview Questions

Q1: Why constructor injection is preferred?

Because it ensures mandatory dependencies and improves testability.

Q2: Is @Autowired mandatory on constructor?

No (if only one constructor exists).

One-Line Summary

Constructor Injection provides dependencies **at the time of object creation**, making the object complete and safe.

Setter Injection

Definition

Setter Injection is a type of **Dependency Injection** where **dependencies are provided using setter methods after object creation**.

Key Points

- Uses setter methods to inject dependencies
- Injection happens **after bean creation**
- Dependencies can be **optional**
- More flexible but less safe than constructor injection

Purpose / Importance

- Useful when dependencies are **not mandatory**
- Allows changing dependencies at runtime

- Simple to understand for beginners

How It Works

1. Spring creates the bean using default constructor
2. Calls setter methods
3. Injects dependencies
4. Bean becomes ready to use

Simple Code Example

```
@Component
public class Car {

    private Engine engine;

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

Real-life Example

TV remote batteries
You can insert or replace batteries **after buying the remote.**

Setter Injection vs Constructor Injection

Feature	**Setter Injection**	**Constructor Injection**
Injection time	After creation	At creation
Mandatory dependency	No	Yes
Immutability	No	Yes
Recommended	No	Yes

Common Interview Questions

Q1: When should setter injection be used?
When dependency is optional.
Q2: Can setter injection cause issues?
Yes, object may be in incomplete state.

One-Line Summary

Setter Injection injects dependencies **after object creation**, suitable for optional dependencies.

Field Injection

Definition

Field Injection is a type of **Dependency Injection** where **Spring injects dependencies directly into class fields** using @Autowired.

Key Points

- No constructor or setter required
- Uses @Autowired on variables
- Quick and less code

- **Not recommended** for large or testable applications

Purpose / Importance

- Easy and fast to write
- Commonly seen in beginner or legacy projects
- Useful for simple demos

How It Works

1. Spring creates the bean
2. Scans fields with @Autowired
3. Injects matching dependency
4. Bean becomes ready

Simple Code Example

```
@Component
public class Car {

    @Autowired
    private Engine engine;
}
```

Real-life Example

Room light switch
Electricity comes directly through wires hidden in the wall.

Field Injection vs Constructor Injection

Feature	Field Injection	Constructor Injection
Boilerplate code	Very less	More
Testability	Poor	Good
Immutability	Poor	Good
Recommended	Poor	Good

Disadvantages (Interview Important)

- Hard to unit test
- Breaks encapsulation
- Dependency is hidden
- Cannot create object manually

Common Interview Questions

Q1: Is field injection recommended?
No, constructor injection is preferred.
Q2: Why field injection is bad for testing?
Dependencies cannot be easily mocked.

One-Line Summary

Field Injection is simple but **not a best practice** in real projects.

@Autowired Annotation

Definition

@Autowired is a Spring annotation used to **automatically inject dependencies** into a bean by **type.**

Key Points

- Performs **automatic dependency injection**
- Works by **type matching**
- Can be used on:
  - Constructor
  - Setter
  - Field
- Provided by Spring Framework

Purpose / Importance

- Eliminates manual object creation
- Reduces boilerplate code
- Makes application loosely coupled

How @Autowired Works

1. Spring scans beans
2. Finds matching type
3. Injects dependency
4. Application runs smoothly

Usage Examples

Constructor Injection (Recommended)

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

Setter Injection

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

Field Injection

```
@Autowired
private Engine engine;
```

Real-life Example

Auto gear car
Gears shift automatically without driver involvement.

@Autowired Resolution Priority

1. By **Type**
2. By **Name** (if conflict)
3. By @Qualifier

Common Problems

Multiple beans of same type

Use @Qualifier

```
@Autowired
@Qualifier("petrolEngine")
private Engine engine;
```

Common Interview Questions

Q1: Is @Autowired mandatory?

No, if only one constructor exists.

Q2: What happens if dependency not found?

Spring throws NoSuchBeanDefinitionException.
Q3: Difference between @Autowired and @Inject?
@Autowired is Spring-specific; @Inject is Java standard.

One-Line Summary
@Autowired lets Spring **automatically inject required dependencies.**

@Qualifier Annotation

Definition
@Qualifier is a Spring annotation used to **resolve ambiguity** when **multiple beans of the same type** exist.

Key Points

- Used **along with @Autowired**
- Matches bean by **name**
- Avoids NoUniqueBeanDefinitionException
- Increases injection accuracy

Purpose / Importance

- Helps Spring decide **which exact bean to inject**
- Essential when multiple implementations are present
- Common in real-world projects

When Do We Need @Qualifier?
When:

- More than one bean of the same type exists
- Spring gets confused during autowiring

Simple Code Example

```
@Component("petrolEngine")
public class PetrolEngine implements Engine { }

@Component("dieselEngine")
public class DieselEngine implements Engine { }

@Component
public class Car {

    @Autowired
    @Qualifier("petrolEngine")
    private Engine engine;
}
```

Real-life Example
Calling a person by **full name** when many people share the same name

---

@Qualifier vs @Primary

Feature	@Qualifier	@Primary
Selection	Explicit	Default
Used at injection	Yes	No

Feature	@Qualifier	@Primary
Priority control	Developer	Spring

Common Interview Questions

Q1: Why do we use @Qualifier?
To select a specific bean when multiple beans exist.

Q2: Can @Qualifier work without @Autowired?
No, it works together with dependency injection.

Q3: What exception occurs without @Qualifier?
NoUniqueBeanDefinitionException.

One-Line Summary
@Qualifier tells Spring **exactly which bean to inject.**

@Primary Annotation

Definition
@Primary is a Spring annotation used to **give higher priority to a bean** when **multiple beans of the same type** exist.

Key Points

- Used on **bean definition**
- Acts as **default choice**
- Works with @Autowired
- Avoids ambiguity without using @Qualifier

Purpose / Importance

- Simplifies dependency injection
- Reduces repeated use of @Qualifier
- Makes configuration cleaner

How @Primary Works

- Spring scans all beans of same type
- Bean marked with @Primary is selected
- Injected automatically

Simple Code Example

```
@Component
@Primary
public class PetrolEngine implements Engine { }

@Component
public class DieselEngine implements Engine { }

@Component
public class Car {

    @Autowired
    private Engine engine; // PetrolEngine injected
}
```

Real-life Example
Default SIM in mobile
Calls use the primary SIM unless you choose another.

@Primary vs @Qualifier

Feature	@Primary	@Qualifier
Priority	Default	Explicit
Location	Bean	Injection point
Flexibility	Less	More
Usage	Simple cases	Complex cases

Common Interview Questions
Q1: Can @Primary and @Qualifier be used together?
Yes, @Qualifier overrides @Primary.
Q2: Where do we use @Primary?
On the bean class or @Bean method.

One-Line Summary
@Primary tells Spring **which bean to choose by default.**

Annotation Configuration (Spring)

Definition
Annotation Configuration is a way of configuring Spring applications using **annotations instead of XML,** making code **cleaner and easier to manage.**

Key Points
- No XML configuration needed
- Uses annotations directly in Java classes
- Easier to read and maintain
- Most commonly used in modern Spring applications

Purpose / Importance
- Reduces boilerplate XML code
- Faster development
- Better readability and productivity

Commonly Used Annotations
Stereotype Annotations
- @Component – Generic bean
- @Service – Business logic
- @Repository – DAO layer
- @Controller / @RestController – Web layer

Dependency Injection
- @Autowired
- @Qualifier
- @Primary

Configuration
- @Configuration
- @Bean
- @ComponentScan

Simple Example

```
@Configuration
@ComponentScan("com.example")
public class AppConfig {
}
@Component
public class Car {
}
```

Spring automatically detects and manages the Car bean.

How Annotation Configuration Works

1. Spring scans packages
2. Finds annotated classes
3. Creates beans
4. Injects dependencies
5. Application runs

Real-life Example

Smartphone apps

No manual setup files — everything auto-configured.

Annotation Configuration vs XML Configuration

Feature	Annotation	XML
Readability	High	Low
Boilerplate	Less	More
Modern usage	Yes	No
Maintenance	Easy	Hard

Common Interview Questions

Q1: Why annotation configuration is preferred?

Because it is cleaner and easier to maintain.

Q2: Which annotation replaces XML bean definition?

@Component and @Bean.

Q3: What is @ComponentScan?

It tells Spring where to scan for beans.

One-Line Summary

Annotation configuration makes Spring **simple, clean, and developer-friendly.**

Java Configuration (Spring)

Definition

Java Configuration is a way of configuring Spring applications using **pure Java classes** instead of XML, by using @Configuration and @Bean.

Key Points

- No XML configuration
- Uses Java classes and methods
- Type-safe and compile-time checked
- Clean and readable configuration

Purpose / Importance

- Avoids XML errors
- Easier refactoring using IDE
- Preferred in modern Spring & Spring Boot projects

Core Annotations Used

- @Configuration – Marks configuration class
- @Bean – Defines a Spring bean
- @ComponentScan – Scans packages for beans

Simple Example

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

    @Bean
    public Engine engine() {
        return new Engine();
    }

    @Bean
    public Car car() {
        return new Car(engine());
    }
}
```

Spring manages Car and Engine as beans.

How Java Configuration Works

1. Spring loads @Configuration class
2. Executes @Bean methods
3. Registers returned objects as beans
4. Injects dependencies

Real-life Example

Home wiring plan
Everything is planned and connected using a blueprint (Java code).

Java Configuration vs Annotation Configuration

Feature	Java Config	Annotation Config
Bean creation	@Bean methods	@Component
Control	More	Less
Use case	Complex setup	Simple setup

Common Interview Questions

Q1: What is @Configuration?

It marks a class as a source of Spring bean definitions.

Q2: Difference between @Bean and @Component?

@Bean is method-level; @Component is class-level.

Q3: Is Java config better than XML?

Yes, it is type-safe and easier to maintain.

One-Line Summary

Java Configuration uses **pure Java code** to configure Spring beans.

@Configuration & @Bean (Spring)

@Configuration

Definition

@Configuration is a Spring annotation used to **mark a class as a configuration class** that defines Spring beans.

Key Points

- Replaces XML configuration
- Contains @Bean methods
- Ensures **singleton behavior** of beans
- Processed by Spring container

Purpose / Importance

- Central place for application configuration
- Clean and type-safe
- Easy to maintain and test

Simple Example

```
@Configuration
public class AppConfig {
}
```

@Bean

Definition

@Bean is used to **define a Spring bean at method level** inside a @Configuration class.

Key Points

- Method return object becomes a Spring bean
- Bean name = method name (default)
- Useful for third-party classes
- Provides fine control over object creation

Simple Example

```
@Configuration
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine();
    }
}
```

How They Work Together

1. Spring loads @Configuration class
2. Executes @Bean methods
3. Registers returned objects as beans
4. Manages lifecycle and dependencies

Real-life Example

Factory blueprint

- @Configuration → factory plan
- @Bean → machines produced

@Configuration vs @Component

Feature	@Configuration	@Component
Purpose	Bean definitions	Generic bean
Contains @Bean	Yes	No
Configuration role	Yes	No

@Bean vs @Component

Feature	@Bean	@Component
Level	Method	Class
Use case	External/3rd party	Custom classes
Control	High	Limited

Common Interview Questions

Q1: Why use @Configuration?

To define Spring beans using Java code.

Q2: Why use @Bean instead of @Component?

When we cannot modify the class (third-party libraries).

Q3: Default bean name for @Bean?

Method name.

One-Line Summary

@Configuration defines **where beans are created,** and @Bean defines **how beans are created.**

@Component & @Service (Spring)

@Component

Definition

@Component is a **generic Spring stereotype annotation** used to mark a class as a **Spring-managed bean.**

Key Points

- Enables component scanning
- Automatically detected by Spring
- Can be used for any class
- Base annotation for other stereotypes

Purpose / Importance

- Reduces XML/Java config
- Lets Spring manage object lifecycle
- Quick and flexible

Simple Example

```
@Component
public class EmailUtil {
}
```

@Service

Definition

@Service is a **specialized form of @Component** used for **business logic layer classes.**

Key Points

- Semantically represents service layer
- Improves code readability
- Supports AOP (transactions, logging)
- Helps in layered architecture

Purpose / Importance

- Clearly separates business logic
- Easy to identify service classes
- Best practice in enterprise apps

Simple Example

```
@Service
public class PaymentService {
}
```

Relationship Between @Component & @Service

- @Service **internally uses @Component**
- Functionally same
- Difference is **meaning and clarity**

Real-life Example

Company structure

- @Component → Any employee
- @Service → Manager handling operations

@Component vs @Service (Interview Favorite)

Feature	@Component	@Service
Type	Generic	Business-specific
Layer	Any	Service layer
Meaning	Neutral	Business logic
Best practice	Okay	Preferred

Common Interview Questions

Q1: Is there any functional difference?

No, only semantic difference.

Q2: Why use @Service instead of @Component?

For better readability and layered design.

Q3: Can @Service be replaced with @Component?

Yes, but not recommended.

One-Line Summary

@Component is generic, while @Service clearly represents **business logic.**

@Repository & @Controller (Spring)

@Repository

Definition

@Repository is a Spring stereotype annotation used to mark a class as a **Data Access Object (DAO).**

Key Points

- Used in **persistence layer**
- Handles database operations
- Enables **exception translation**
- Internally a @Component

Purpose / Importance

- Clean separation of data access logic
- Converts database exceptions into Spring exceptions
- Improves readability and maintainability

Simple Example

```
@Repository
public class UserRepository {
}
```

@Controller

Definition

@Controller is a Spring annotation used to handle **web requests** and return **views.**

Key Points

- Used in **Spring MVC**
- Handles HTTP requests
- Returns view name (JSP/HTML)
- Works with @RequestMapping

Purpose / Importance

- Acts as **entry point** for web applications
- Separates request handling from business logic
- Supports MVC architecture

Simple Example

```
@Controller
public class UserController {

    @RequestMapping("/users")
    public String getUsers() {
        return "users";
    }
}
```

Real-life Example

Restaurant system

@Controller → Waiter (takes orders)

- @Repository → Store room (fetches ingredients)

@Repository vs @Controller (Interview Table)

Feature	@Repository	@Controller
Layer	DAO	Web
Handles	DB operations	HTTP requests
Returns	Data	View
MVC role	Model	Controller

Common Interview Questions

Q1: Why use @Repository instead of @Component?

For DAO clarity and exception handling.

Q2: Can @Controller return data?

No, it returns view names (use @RestController for data).

Q3: Are these stereotype annotations?

Yes.

One-Line Summary

@Repository talks to the database, and @Controller talks to the client.

Bean Scopes

Singleton & Prototype

Singleton Scope

Definition

Singleton is the **default Spring bean scope**, where **only one instance of a bean is created per Spring container**.

Key Points

- One object shared across application
- Created at container startup (eager by default)
- Same instance returned every time
- Default scope in Spring

Purpose / Importance

- Saves memory
- Suitable for **stateless beans**
- Commonly used in service and DAO layers

Simple Example

```
@Component
@Scope("singleton")
public class UserService {
}
```

Real-life Example

Water tank

One tank supplies water to all houses.

Prototype Scope

Definition

Prototype scope creates a **new bean instance every time it is requested** from the container.

Key Points

- Multiple objects created
- Created only when requested
- Spring does not manage full lifecycle
- Used for **stateful beans**

Purpose / Importance

- When each user/process needs a separate object
- Avoids shared state issues

Simple Example

```
@Component
@Scope("prototype")
public class UserSession {
}
```

Real-life Example

Disposable cups
New cup for every customer.

Singleton vs Prototype (Interview Favorite)

Feature	Singleton	Prototype
Instances	One	Many
Default	Yes	No
Memory	Low	Higher
Lifecycle	Fully managed	Partially managed
Use case	Stateless	Stateful

Important Interview Notes

- Spring default scope = **Singleton**
- Prototype beans are **not destroyed by Spring**
- Singleton ≠ Java Singleton pattern

Common Interview Questions

Q1: What is default scope in Spring?
Singleton.
Q2: When to use prototype?
When bean is stateful.
Q3: Does Spring manage prototype bean destruction?
No.

One-Line Memory Trick

Singleton = One object for all, Prototype = New object for each request

---

Web Bean Scopes
Request | Session | Application

- These scopes are mainly used in **Spring Web / Spring MVC applications.**

---

1. Request Scope

Definition

A Request-scoped bean is created **once per HTTP request** and destroyed after the request is completed.

Key Points

- New bean for each request
- Lives only during one HTTP request
- Not shared between users
- Used in web apps only

Purpose / Importance

- Useful for request-specific data
- Avoids data leakage between requests

Simple Example

```
@Component
@Scope("request")
public class RequestData {
}
```

Real-life Example

Online form fill
Data exists only until you submit the form.

---

2. Session Scope

Definition

A Session-scoped bean is created **once per user session** and lives until the session expires.

Key Points

- One bean per user
- Shared across multiple requests of same user
- Destroyed when session ends

Purpose / Importance

- Stores user-specific data
- Common in login-based apps

Simple Example

```
@Component
@Scope("session")
public class UserSession {
}
```

Real-life Example

Shopping cart
Items remain until logout.

---

3. Application Scope

Definition

An Application-scoped bean has **one instance for the entire web application.**

Key Points

- Shared across all users
- Created at application startup
- Destroyed when application stops

Purpose / Importance

- Used for global data
- Saves memory

Simple Example

```
@Component
@Scope("application")
public class AppConfigData {
}
```

Real-life Example

Company notice board
Same information for everyone.

Comparison Table (Very Important for Interview)

Scope	Instance	Lifetime	Shared
Request	Per request	One HTTP request	No
Session	Per user	User session	No
Application	One	App lifetime	Yes

Important Interview Notes

- These scopes work only in **web applications**
- Not applicable in standalone Spring Core apps
- Need **Spring MVC / Spring Boot Web**

Common Interview Questions

Q1: Which scope is used for login details?
Session scope.
Q2: Which scope is used for form data?
Request scope.
Q3: Which scope is shared among all users?
Application scope.

One-Line Summary

Request = One request, Session = One user, Application = One app

---

Bean Initialization & Destruction (Spring)

1. Bean Initialization

Definition

Initialization is the **phase where a Spring bean is prepared for use** after dependencies are injected.

Key Points

- Happens **after dependency injection**
- Used to perform setup logic
- Bean becomes ready for business use

Ways to Do Initialization

1. Using @PostConstruct

```
@PostConstruct
public void init() {
    System.out.println("Bean initialized");
}
```

2. Using init-method (XML / Java Config)

```
@Bean(initMethod = "init")
public Car car() {
    return new Car();
}
```

Real-life Example

Washing machine setup
Machine is installed and checked before use.

---

2. Bean Destruction

Definition

Destruction is the **phase where cleanup activities are performed** before the bean is removed.

Key Points

- Happens when container shuts down
- Used to release resources
- Not called for prototype beans

Ways to Do Destruction

1. Using @PreDestroy

```
@PreDestroy
public void destroy() {
    System.out.println("Bean destroyed");
}
```

2. Using destroy-method

```
@Bean(destroyMethod = "destroy")
public Car car() {
    return new Car();
}
```

Real-life Example

Closing office
Lights off, systems shut down.

Bean Lifecycle Flow (Easy to Remember)

1. Bean Instantiation
2. Dependency Injection
3. Initialization
4. Bean Ready
5. Destruction

Important Interview Notes

- Initialization runs **once per bean**
- Destruction not called for **prototype scope**
- @PostConstruct runs before business methods
- @PreDestroy runs at container shutdown

Common Interview Questions

Q1: When does initialization happen?
After dependency injection.
Q2: When is destroy method called?
When container is closed.
Q3: Is destroy called for prototype beans?
No.

One-Line Summary
Initialization prepares the bean for use, and destruction cleans it up.

@PostConstruct & @PreDestroy (Spring)

@PostConstruct

Definition

@PostConstruct is used to define a **method that runs after dependency injection is completed.**

Key Points

- Called **after bean creation**
- Runs only **once**
- Used for initialization logic
- Comes from jakarta.annotation

Purpose / Importance

- Perform setup tasks
- Validate injected dependencies
- Initialize resources

Simple Example

```
@PostConstruct
public void init() {
    System.out.println("Initialization logic");
}
```

Real-life Example

Starting a laptop
System checks run before use.

---

2. @PreDestroy

Definition

@PreDestroy is used to define a **method that runs before the bean is destroyed.**

Key Points

- Called during container shutdown
- Used for cleanup activities

- Not executed for prototype beans
- Runs only once

Purpose / Importance

- Close DB connections
- Release resources
- Save application state

Simple Example

```
@PreDestroy
public void cleanup() {
    System.out.println("Cleanup logic");
}
```

Real-life Example

Shutting down a computer
Programs are closed safely.

@PostConstruct vs @PreDestroy (Interview Table)

Feature	@PostConstruct	@PreDestroy
Called	After DI	Before destruction
Purpose	Initialization	Cleanup
Runs	Once	Once
Prototype beans	Yes	No

Important Interview Notes

- Works only for **singleton & web scopes**
- Part of **JSR-250**
- Runs automatically (no need to call)

Common Interview Questions

Q1: When is @PostConstruct executed?
After dependency injection.
Q2: When is @PreDestroy executed?
Before container shutdown.
Q3: Is @PreDestroy called for prototype beans?
No.

One-Line Summary

@PostConstruct prepares the bean, @PreDestroy cleans it up.

@PropertySource

Definition

@PropertySource is a Spring annotation used to **load external .properties files** into the Spring application context.

Key Points

- Used on @Configuration classes
- Supports multiple property files
- Works with @Value or Environment to inject values

Purpose / Importance

- Externalizes configuration (DB URLs, usernames, passwords)
- Avoids hardcoding values in Java classes
- Makes apps **flexible and maintainable**

Syntax / Example

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
}
@Value("${db.url}")
private String dbUrl;
```

Real-life Example

Like keeping **settings in a separate settings file** instead of hardcoding them inside a program.
Example: TV remote settings stored in a manual instead of setting it manually every time.

Interview Tips / Questions

Q: Can Spring Boot use @PropertySource?
A: Usually not needed; Spring Boot auto-loads application.properties.
Q: How do you read a value from a properties file?
A: Using @Value("${property.key}") or Environment.getProperty("property.key").
Q: Can we use multiple property files?
A: Yes, by adding multiple @PropertySource annotations or using @PropertySources.

One-Line Summary

@PropertySource lets Spring **load external configuration files** for flexible and maintainable applications.

@Value Annotation

Definition

@Value is a Spring annotation used to **inject values into Spring beans** from **properties files, system properties, or literals**.

Key Points

- Can inject values into **fields, setters, or constructor parameters**
- Supports ${property.key} syntax for properties files
- Can provide **default values** if the key is missing

Purpose / Importance

- Reads external configuration easily
- Avoids hardcoding values in the code
- Makes applications **flexible and maintainable**

Syntax / Example

Injecting from properties file:

```
@Value("${db.username}")
private String username;
```

Injecting a default value:

```
@Value("${server.port:8080}")
private int port;
```

Injecting a literal value:

```
@Value("SpringCore")
private String courseName;
```

Real-life Example

Like **reading the electricity bill amount** from a bill instead of guessing it—dynamic and accurate.

Interview Tips / Questions

- **Q:** Can @Value read from application.properties?
  A: Yes, using ${key} syntax.
- **Q:** Difference between @Value and @Autowired?
  A: @Value injects simple values; @Autowired injects beans (objects).
- **Q:** Can you set a default value in @Value?
  A: Yes, using : syntax like ${key:defaultValue}.

One-Line Summary

@Value injects **values from properties files or literals** into Spring beans for dynamic configuration.