# Spring Core & Spring Boot

## Fundamentals

# What's Inside

01

# Tight Coupling

*The fundamental problem Spring was built to solve*

## What is Tight Coupling?

Tight coupling occurs when one class is permanently bound to another — making it impossible to change, replace, or test independently. Two main causes:

• The 'new' keyword — class OrderService creates class EmailNotification internally using new EmailNotification(). OrderService decides for itself what to use, hardcoded into source code.

• Inheritance (extends): class OrderService extends class EmailNotification. OrderService inherits ALL of EmailNotification's behavior. If EmailNotification changes, OrderService breaks. OrderService can never switch to a different parent. The bond is permanent and compile-time locked.

## Why is it a problem?

1. No flexibility — want SMS instead of Email? You must open and rewrite the class.

2. No testability — can't test OrderService without creating a real EmailNotification.

3. Violates Open/Closed Principle — classes should be open for extension but closed for modification.

4. Ripple effect — if EmailNotification's method changes, every class using it breaks.

**Real-world: PaymentService creates new StripeGateway() — can never switch to PayPal. UserDAO hardcodes new MySQLConnection() — can't migrate to PostgreSQL.**

**Ex: OrderService locked to EmailNotification:**

```
package com.vk.tight;

public class OrderService {

    // TIGHT COUPLING: OrderService directly creates
    // EmailNotification using 'new'
    // It decided for ITSELF what notification to use
    EmailNotification notif = new EmailNotification();

    public void placeOrder(String item) {
        System.out.println("Order placed: " + item);
        notif.send("Order for " + item + " confirmed!");
    }
}
```

### What's wrong with this code?

OrderServicehas ONE job—processing orders. But it's also deciding HOW notifications are sent. It chose email. Tomorrow the client wants SMS — you must open OrderService.java and change it. That's a violation of the Single Responsibility Principle.

The 'new' keyword is the culprit. Every time you write 'new SomeClass()' inside another class, you're creating a tight coupling.

**Ex:**

```
package com.tight;

public class OrderService extends EmailNotification{

    // TIGHT COUPLING via INHERITANCE: OrderService IS-A EmailNotification
    //OrderService is permanently locked to EmailNotification through 'extends'

    public void placeOrder(String item) {
        System.out.println("Order placed for: " + item);
        // send() comes from the parent class EmailNotification
        send("Your order for " + item + " has been confirmed!");
    }
}
```

### What's wrong with this code?

OrderService becomes tightly bound to EmailNotification, even though it should only use it, not be it.

Switching to another notification type like SMS requires changing inheritance or rewritingthe class.

Java supports single inheritance, so OrderService cannot extend any other useful class.

Any change in EmailNotification directly impacts OrderService, increasing dependency.

It’s impossible to support multiple notifications (email + SMS) through inheritance.

02

# Loose Coupling + DI

*The solution that makes Spring possible*

## What is Loose Coupling?

Loose coupling is a design principle where classes depend on abstractions (interfaces) instead of concrete implementations. The class knows WHAT it needs (a notification service) but not HOW it's implemented (email, SMS, or WhatsApp).

The interface acts as a contract — it defines the method signatures that any implementation must follow. As long as the contract is honored, you can swap implementations freely without changing a single line in the dependent class.

This is powered by runtime polymorphism — the ability to reference different objects through the same interface type.

## What is Dependency Injection (DI)?

Dependency Injection is the mechanism that makes loose coupling work in practice. Instead of a class creating its own dependencies internally (using 'new'), someone OUTSIDE the class provides them.

Think of it as food delivery: instead of cooking your own meal (creating dependencies), someone delivers it to you (injects dependencies). The class just declares what it needs.

Two delivery methods:
- Constructor Injection — dependency delivered at birth (object creation)
- Setter Injection — dependency delivered after birth (via a setter method)

Ex:

```
package com.vk.loose;

// The CONTRACT — any class implementing this
// MUST have a send() method
public interface NotificationService {
    void send(String message);
}
```

```
public class OrderService {
    private NotificationService notif; // INTERFACE!

    //CONSTRUCTOR INJECTION — mandatory dependency
    public OrderService(NotificationService notif) {
        this.notif = notif;
    }

    //SETTER INJECTION — swappable at runtime
    public void setNotification(NotificationService n){
        this.notif = n;
    }

    public void placeOrder(String item) {
        notif.send("Order for " + item);
    }
}
```

**DI is happening, but main() does the wiring manually. With 500 classes, you can't wire everything by hand. This is exactly what Spring's IoC container solves.**

## Constructor vs Setter Injection

### Constructor Injection

Dependency passed through the constructor at object creation time. The object cannot exist without it — it's mandatory. The field can be declared 'final', making the object truly immutable after creation. The Spring team officially recommends this for all mandatory dependencies.

### Setter Injection

Dependency passed through a setter method after object creation. The object CAN exist without it — making the dependency optional. Allows changing the implementation at runtime by calling the setter again. But the object may be in an incomplete state between creation and the setter call, risking NullPointerException.

| Aspect | Constructor | Setter |
|---|---|---|
| Required? | Mandatory | Optional |
| **Supports final?** | Yes ✓ | No ✗ |
| Swap at runtime? | No | Yes ✓ |
| Null safety | Guaranteed non-null | Possible NPE |
| Spring says | **PREFERRED** ✓ | Optional deps only |

03

# Spring IoC Container

*Spring takes over object creation and wiring*

## What is inversion of Control(IoC)?

Inversion of Control is a fundamental design principle where the control of the object creation and dependency wiring is inverted – transferred FROM your code To the framework.

In traditional programming, YOUR code creates objects, decides which implementations to use, and wires everything together. With IoC, you define a blueprint (which beans exist, how they connect) and the framework does all the heavy lifting.

You go from being the architect AND the construction crew to being ONLY the architect. Spring is your construction crew — it reads your blueprint and builds everything according to your specifications.

## What is a Bean?

A Bean is simply a Spring-managed object. The distinction matters:

• When YOU create an object using 'new' → it's just a regular Java object. You control its entire lifecycle. Spring knows nothing about it.

• When SPRING creates an object → it becomes a Bean. Spring controls its creation, dependency injection, initialization, and destruction. It lives inside Spring's IoC container.

The IoC container (ApplicationContext) is the central registry where all beans live. When you need a bean, you ask the container — you never use 'new' for managed objects again.

**Ex: BEFORE (you control everything):**

```
// YOU are the construction crew
// YOU create every object, wire every dependency
EmailNotification email = new EmailNotification();
OrderService svc = new OrderService(email);
svc.placeOrder("Spring Course");
```

### AFTER (Spring controls everything):

```
// SPRING is the construction crew
// You just ask for what you need
ApplicationContext ctx =
new ClassPathXmlApplicationContext("config.xml");

OrderService svc = ctx.getBean(OrderService.class);
svc.placeOrder("Spring Course");
// Spring created OrderService AND injected
// EmailNotification into it — automatically.
```

The key shift: Your code never uses 'new' for managed objects again. You define the blueprint (XML or annotations), Spring builds everything.

## BeanFactory vs ApplicationContext

**Two types of IoC containers exist in Spring:**

BeanFactory is the basic, lightweight container. It uses lazy loading — beans are created only when you explicitly request them. Saves memory but discovers problems late.

ApplicationContext extends BeanFactory and adds enterprise features. Uses eager loading — ALL singleton beans are created at startup. Catches errors early and provides additional capabilities.

| Feature | BeanFactory | ApplicationContext |
|---|---|---|
| Loading | Lazy — on getBean() | **Eager — ALL at startup** |
| i18n support | No | Yes — MessageSource |
| Event publishing | No | Yes — ApplicationEvent |
| AOP integration | No | Yes — proxies, aspects |
| Annotation processing | Manual setup | Automatic |
| Use when | Resource-constrained (rare) | **99% of all applications** |

**ApplicationContext inherits everything from BeanFactory plus all features above. Always use ApplicationContext unless you have a specific reason not to.**

04

# Xml Configuration

*The original way to define beans in Spring*

### How XML Configuration Works

In early Spring, you defineevery beanin an XML file. Each <bean> tag tells Spring: create an instance of this class with this name. The <property> and <constructor-arg> tags tell Spring how to wire dependencies.

Spring reads this XML at startup, creates all beans, and injects all dependencies according to your instructions. The XML file IS your blueprint.

```
<!-- Define beans: Spring calls constructors -->
<bean id="email"
      class="com.vk.notification.EmailNotification"/>

<!-- Setter injection: Spring calls setNotification() -->
<bean id="orderService"
      class="com.vk.order.OrderService">
    <property name="notification" ref="email"/>
</bean>

<!-- Constructor injection alternative -->
<bean id="orderService2"
      class="com.vk.order.OrderService">
    <constructor-arg ref="email"/>
</bean>
```

| Attribute | Purpose | Example |
|---|---|---|
| ref | References another bean | ref="email" |
| value | Primitive/String value | value="Rohan" |

## P & C Namespace Shortcuts

### What are P and C namespaces?

AsXML configurationsgrow,writing <property> and <constructor-arg> tags for every field becomes verbose. P-namespace and C-namespace are XML shortcuts that let you define thesame injection ina singleline attribute.

**Verbose — 3 separate property tags:**

```
<bean id="emp" class="com.vk.Employee">
    <property name="empid" value="1"/>
    <property name="name" value="Rohan"/>
    <property name="depart" ref="depart"/>
</bean>
```

### P-namespace — same thing, one line:

```
<bean id="emp" class="com.vk.Employee"
p:empid="1" p:name="Rohan"
p:depart-ref="depart"/>
```

| Syntax | Type | Reference? |
|---|---|---|
| `p:field="value"` | Setter injection | No — primitive |
| `p:field-ref="beanId"` | Setter injection | **Yes — bean ref** |
| `c:param="value"` | Constructor injection | No — primitive |
| `c:param-ref="beanId"` | Constructor injection | **Yes — bean ref** |

05

# Xml Configuration

*Let Spring figure out the wiring automatically*

### What is Autowiring?

Instead of manually specifying every dependency with <property> or <constructor-arg>, you tell Spring to figure out the wiring automatically. Spring matches beans to dependencies based on their name, type, or constructor parameters.

This reduces XML verbosity but introduces the risk of unexpected behavior — Spring may wire the wrong bean or fail silently.

| Mode | How Spring Matches | Risk |
|---|---|---|
| *no* | Manual — you specify each dep | None — safest |
| *byName* | Bean id matches setter name | Rename → silent null! |
| *byType* | Bean TYPE matches setter type | 2 beans → exception |
| *constructor* | Bean TYPE matches constructor | Same as byType |

### Resolving Ambiguity

When two beans of the same type exist (Email + SMS both implement NotificationService), byType fails with NoUniqueBeanDefinitionException. Two fixes:

- primary="true" — marks one bean as the default
- autowire-candidate="false" — excludes a bean from autowiring

06

# Stereotypes + Scanning

*Annotations replace dozens of XML bean tags*

## What is Component scanning?

Instead of writing a <bean> tag inXML for every class, you annotate the class itself with a stereotype annotation (like @Service or @Component) and tell Spring to scan a package. Spring discovers all annotated classes and registers them as beans.

One line of XML — <context:component-scan base-package="com.vk"/> — replaces potentially hundreds of <bean> tags. This is the shift from XML-driven to annotation-driven configuration.

## What are Stereotype Annotations?

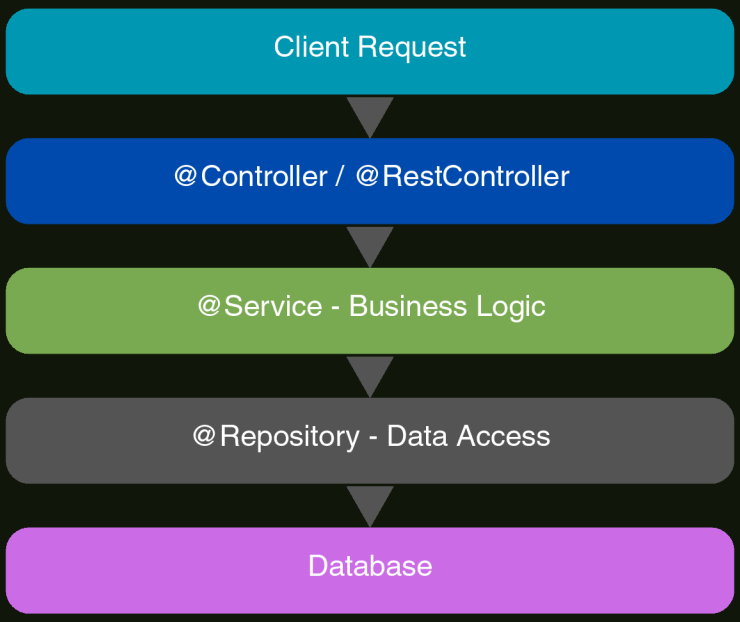
Stereotypeannotationsmark classes withtheir architectural role. They are ALL @Component underneath — open the source of @Service and you'll see @Component on it. The specialized names exist for:

- Readability — instantly know a class's purpose
- Layer separation — enforce architectural boundaries
- Special behaviors — @Repository auto-translates DB exceptions, @Controller enables HTTP request mapping

Default bean name rule: class name with first letter lowercased. OrderService → "orderService". Custom: @Service("myName").

## Stereotype Hierarchy & Layered Architecture

| Annotation | Layer | Special Behavior |
|---|---|---|
| `@Component` | Generic | Base — parent of all below |
| `@Service` | Business logic | Semantic clarity only |
| `@Repository` | Data access | **Auto DB exception translation** |
| `@Controller` | Web / HTTP | Enables @RequestMapping |
| `@RestController` | REST API | @Controller + @ResponseBody |
| `@Configuration` | Config | Holds @Bean definitions |




07

# @Autowired - 3 Styles

*Annotation - based dependency injection*

## What is @Autowired?

@Autowired tells Spring: "automatically inject the matching bean here." Spring looks at the type and finds a matching bean. If exactly one matches, it's injected. If zero match, it fails. If two+ match, you need @Qualifier or @Primary to resolve ambiguity.

You can place @Autowired in three locations, each with different trade-offs. The Spring team recommends: always use constructor injection for mandatory dependencies.

## ①Field Injection — NOT recommended

Shortest syntax but hides dependencies. Can't make field 'final' (no immutability), testing requires reflection to inject mocks. Class's API doesn't reveal what it needs.

## ② Constructor Injection — RECOMMENDED

Dependencies explicit in the constructor. Supports 'final' fields (true immutability). Easy to test — just pass mocks. If only ONE constructor exists, @Autowired is optional (Spring 4.3+).

## ③Setter Injection — for optional deps

Use when a dependency is optional — the object can function without it. Allows swapping at runtime. Risk: NullPointerException if used before setter is called.

# @Primary, @Qualifier — Example

```
package com.vk.notification;

@Service
@Primary  //default choice when multiple match
public class EmailNotification
    implementsNotificationService {
    publicvoidsend(String m) {
        System.out.println("[EMAIL] " + m);
    }
}

@Service // bean name = "smsNotification"
public class SmsNotification
    implements NotificationService { ... }
```

```
// @Qualifier OVERRIDES @Primary — picks exact bean
@Autowired
@Qualifier("smsNotification")
private NotificationService notif; // SMS wins!
```

| Priority | Mechanism | Behavior |
|---|---|---|
| **1st (highest)** | `@Qualifier` | Always wins — picks by name |
| 2nd | `@Primary` | Default when no @Qualifier |
| 3rd (lowest) | By type | Fails if 2+ beans match |

# 08 @Bean vs @Component

*When to use each — and why they both exist*

### @Component (and children: @Service, @Repository, @Controller)

You annotate YOUR class directly. Spring discovers it through component scanning. Use this whenever you own the source code.

The bean name defaults to class name with first letter lower cased: GreetingService → "greetingService". Spring calls the constructor to create the instance, then processes @Autowired.

### @Bean

You write a METHOD inside a @Configuration class that creates and returns an object. Spring calls this method and registers the returned object as a bean.

Use when you DON'T own the source code — you can't add @Component to java.time.LocalTime, javax.sql.DataSource, or any third-party library. Also use when you need custom construction logic (like passing "SHA" to a Password constructor).

The bean name defaults to the method name. @Configuration itself is a @Component — Spring creates a CGLIB proxy to ensure @Bean methods return singletons.

**Rule of thumb: @Component for YOUR classes. @Bean for classes you don't own or need special construction logic.**

## @Bean vs @Component — Example

```
// @Component — you OWN GreetingService
@Service
public class GreetingService {
    @Autowired
    private LocalTime time; // injected by Spring

    public String greet() {
        if (time.getHour() < 12) return "Morning!";
        if (time.getHour() < 17) return "Afternoon!";
        return "Evening!";
    }
}
```

```java
// @Bean — you CAN'T add @Component to LocalTime
@Configuration
public class AppConfig {

    @Bean  // bean name = "createTime"
     publicLocalTime createTime() {
         return LocalTime.now();
    }

    @Bean
    public Password createPasswordBean() {
        return new Password("SHA"); // custom logic
    }
}
```

| Aspect | @Component | @Bean |
|---|---|---|
| Applied to | **Class** | **Method in @Configuration** |

## 09 4 Config Approaches

*The evolution from XML-heavy to auto-configured*

### Why does Spring have 4 configuration styles?

Spring evolved over 20+ years. Each approach reduced boilerplate while keeping backward compatibility:

1. Pure XML (2004) — Everything in XML. Verbose but explicit. No compile-time checking.

2. XML + Annotations (2007) — Beans defined by annotations. XML only for component-scan. Huge reduction.

3. Java Config / Zero XML (2009) — @Configuration class replaces XML entirely. @ComponentScan replaces <context:component-scan>. Full compile-time safety.

4. Spring Boot (2014) — @SpringBootApplication combines @Configuration + @ComponentScan + @EnableAutoConfiguration. Auto-detects classpath dependencies and configures beans. Embedded Tomcat. Properties file for config. 96% annotation-driven — XML essentially gone.

| Style | XML | Annotations | Example |
|---|---|---|---|
| Pure XML | Heavy | None | Ex04 |
| XML + Ann | Minimal | Most beans | Ex07 |
| Java Config | None ✓ | All beans | Ex10 |
| Spring Boot | None ✓ | **Auto-configured** | Ex13 |

# @SpringBootApplication — Example

**One annotation replaces three**

@SpringBootApplication is a convenience annotation that combines @Configuration (this class is a bean source), @ComponentScan (scan current package + sub-packages), and @EnableAutoConfiguration (detect classpath and auto-configure beans). SpringApplication.run() creates ApplicationContext + starts embedded Tomcat.

```
package com.vk.Ex13SpringBootAutoConfig;

@SpringBootApplication
// = @Configuration
//   + @ComponentScan
//   + @EnableAutoConfiguration
public class Ex13...Application {
    public static void main(String[] args) {
        var ctx = SpringApplication.run(
           Ex13...Application.class, args);

        System.out.println(
            ctx.getBeanDefinitionCount());
        //52+ beans! You only wrote 3 classes!
    }
}
```

**SpringApplication.run() does everything: creates ApplicationContext, scans for components, auto-configures beans from classpath, starts embedded Tomcat, and returns the context.**

10

# Bean Lifecycle

*Complete execution order—birth to death*

### What is Bean Lifecycle?

Every Spring bean goesthrough a predictable sequence of events from creation to destruction. Understanding this order is critical because you often need initialization code (load config, open connections) AFTER dependencies are injected, and cleanup code (close connections, flush caches) BEFORE the bean is destroyed.

Spring provides hooks at key points: @PostConstruct for initialization and @PreDestroy for cleanup. These run automatically — you just annotate your methods.

1. **Static block**
   JVM loads class — before Spring
2. **Instance block**
   Java instance initializer
3. **Constructor**
   Spring instantiates the bean
4. **Dependency Injection**
   @Autowired / setter / constructor
5. **@PostConstruct**
   Init logic AFTER all DI complete
★ **BEAN IS READY**
   Business methods execute
6. **@PreDestroy**
   Cleanup BEFORE container shutdown
7. **Garbage collected**
   Bean removed from memory

# @PostConstruct & @PreDestroy — Code

### @PostConstruct

Runs ONCEafterall dependencies injected. Use for loading config, opening connections, validating data. From jakarta.annotation package.

### @PreDestroy

Runs whenApplicationContext shuts down. Use for closing DB connections, flushing caches. Console apps MUST call ctx.close() for this to fire!

```java
package com.vk.service;

@Service
public class ReportService {
    @Autowired
    private DatabaseService db;

    @PostConstruct
    public void init() {
        System.out.println("Init — loading config");
        // validate, open connections, load data
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Cleanup — closing...");
        // close connections, flush caches
    }
}
```

| Mechanism | Annotation | XML Equivalent |
|---|---|---|
| Init | @PostConstruct | init-method="init" |
| Destroy | @PreDestroy | destroy-method="cleanup" |

# 11 Bean Scopes

*How many instances? How long do they live?*

## What is Bean Scope?

Bean scope controls HOW MANY instances of a bean Spring creates and HOW LONG they live. This determines whether every consumer gets the same shared object or a fresh copy.

By default, Spring creates exactly ONE instance of every bean (singleton) and shares it across the entire application. Sometimes you need a fresh instance — like a shopping cart unique per user. That's where scopes come in.

## Singleton (default) — ONE instance, shared everywhere

Spring creates one instance at startup(eager loading) and returns that same instance every time. Perfect for stateless services like OrderService or any utility class. Spring manages full lifecycle including @PreDestroy.

## Prototype — NEW instance every time

Spring creates a brand new instance every time getBean() is called. After creation, Spring forgets about it — @PreDestroy will NOT fire. Use for stateful objects like shopping carts or request-specific data objects.

```
// From Ex12BeanScopes:
@Component
@Scope("prototype") // new instance every time
public class ShoppingCart { /* user-specific */ }

ShoppingCart c1 = ctx.getBean(ShoppingCart.class);
ShoppingCart c2 = ctx.getBean(ShoppingCart.class);
System.out.println(c1 == c2); // FALSE! // Different
hashCodes — different objects
```

| Scope | One Instance Per | Use Case |
|---|---|---|
| *singleton* | Entire container (default) | Services, repos, utilities |
| *prototype* | Every getBean() call | Stateful, shopping carts |
| request | Each HTTP request | Form processing |
| session | Each HTTP session | User auth, preferences |
| application | Entire ServletContext | App-wide config |

## @Lazy — Delay Bean Creation

Singleton beans are eager (created at startup even if unused). @Lazy delays creation until first request. Use for expensive beans that may not always be needed.

12

# Auto - Configuration

*Spring Boot's magic explained*

### What is Auto-Configuration?

Auto-configuration is SpringBoot's signature feature. It examines your classpath — what dependencies are in your pom.xml — and automatically creates and configures beans.

Add spring-boot-starter-web? Boot auto-configures DispatcherServlet, embedded Tomcat, and JSON converters. Add spring-boot-starter-data-jpa? It configures DataSource, EntityManagerFactory, and HikariCP. Add nothing extra? It STILL configures 50+ infrastructure beans.

You write zero bean definitions for standard setups. Override only when needed.

### How it works under the hood

1. @EnableAutoConfiguration(inside@SpringBootApplication) triggers the process.

2. Spring Boot reads META-INF/AutoConfiguration.imports — a list of config classes.

3. Each auto-config class has conditions:
- @ConditionalOnClass — is this class on the classpath?
- @ConditionalOnMissingBean — have you defined your own bean?
- @ConditionalOnProperty — is this property set?

4. If ALL conditions met → beans created. If any fails → silently skipped.
5. Override any auto-config by defining your own @Bean of the same type — Spring Boot backs off thanks to @ConditionalOnMissingBean.

## Proving & Debugging Auto-Config

### Counting auto-created beans:

```
var ctx = SpringApplication.run(App.class, args);
System.out.println(ctx.getBeanDefinitionCount());
// Output: 52+ beans — you only wrote 3 classes!

 for (String n : ctx.getBeanDefinitionNames())
      System.out.println(n); // see every bean
```

### Debug — see what got auto-configured and why:

```
# application.properties
debug=true
# Console shows:
  # Positive matches — what WAS configured
  # Negative matches — what was NOT and why
```

### application.properties — central configuration:

```
spring.application.name=MyApp
server.port=8080
logging.level.org.springframework=DEBUG
# Inject with: @Value("${server.port}")
```

**HikariCP: Spring Boot auto-configures a connection pool. Reuses DB connections instead of opening/closing each time. Zero config needed.**

## The Learning Journey

| Example | Topic | Key Concept |
|---|---|---|
| *Ex01* | **Tight Coupling** | new keyword locks classes |
| *Ex02* | **Tight (extends)** | Inheritance coupling |
| *Ex03* | **Loose Coupling + DI** | Interface + constructor/setter |
| *Ex04* | **Spring IoC** | ApplicationContext takes over |
| *Ex05* | **XML Config** | Setter/constructor, p/c-namespace |
| *Ex06* | **XML Autowiring** | 4 modes: no, byName, byType, constructor |
| *Ex07* | **Stereotypes** | @Service @Repository @Component |
| *Ex08* | **@Autowired** | 3 injection styles + @Qualifier @Primary |
| *Ex09* | **@Bean vs @Component** | Third-party classes, @Configuration |
| *Ex10* | **3 Config Approaches** | Pure XML → XML+Ann → Java Config |
| *Ex11* | **Bean Lifecycle** | @PostConstruct @PreDestroy, full order |
| *Ex12* | **Bean Scopes** | Singleton vs Prototype, @Lazy |
| *Ex13* | **Spring Boot** | @SpringBootApplication, 52+ auto beans |

# Cheat Sheet — Injection & Resolution

| Injection Style | Syntax | When to Use |
|---|---|---|
| **Field** | `@Autowired private X x;` | Never — not recommended |
| **Constructor** | `public Svc(X x) {}` | Always — mandatory deps ✓ |
| **Setter** | `@Autowired setX(X x)` | Optional dependencies only |

| Priority | Mechanism | Behavior |
|---|---|---|
| **1st** | `@Qualifier` | Always wins — picks exact bean by name |
| 2nd | `@Primary` | Default when no @Qualifier specified |
| 3rd | By type | Fails with 2+ beans of same type |

| Scope | Instances | @PreDestroy? | Use For |
|---|---|---|---|
| *`Singleton`* | ONE — shared | Yes ✓ | Stateless services |
| *`Prototype`* | NEW every call | No ✗ | Stateful objects |

| Config Style | XML | Annotations | Example |
|---|---|---|---|
| Pure XML | Heavy | None | `Ex04` |

## Bean Lifecycle — Complete Order:


## Auto-Configuration — How It Works:

### The Process

1. @EnableAutoConfiguration triggers scanning 2. Reads AutoConfiguration.imports for config classes 3. Each class checks conditions: @ConditionalOnClass, @ConditionalOnMissingBean

4. Conditions met → beans created. Not met →skipped.

5. Define your own @Bean of same type →Boot backs off

### Key Annotations Summary:

| Annotation | Purpose |
|---|---|
| `@SpringBootApplication` | @Config + @ComponentScan + @EnableAutoConfig |
| `@Autowired` | Inject matching bean automatically |
| `@Qualifier("name")` | Pick specific bean when multiple match |
| `@Primary` | Default bean when no @Qualifier |
| `@PostConstruct / @PreDestroy` | Lifecycle hooks for init and cleanup |

## 5 Lessons from Implementations

1

**Spring doesn't replace Java — it enhances it**

You still need OOP, interfaces, polymorphism. Spring builds on top of solid Java fundamentals.

2

**Every @Autowired hides a design decision**

Constructor=mandatory.Setter=optional.Field=hidden.Choose deliberately, not lazily.

3

**XML → Annotations → Boot is an evolution, not a replacement**

Understanding XML helps you debug auto-config. The old way teaches WHY the new way exists.

4

**Bean lifecycle matters more than you think**

@PostConstruct forsetup,@PreDestroyforcleanup.Miss these and you'll have resource leaks in production.

5

**Write code — reading is not learning**

while writing code errors occured taught me more than the correct code ever did.

## Tools & Setup

**Java 17**
Language & runtime

**Spring Framework 7.0.4**
Core IoC + DI container

**Spring Boot 3.4.x**
Auto-configuration+embedded Tomcat

**Maven**
Dependencymanagement + builds

**Eclipse IDE**
Development environment

**Git + GitHub**
Version control + code sharing

# What's Next?

**Spring MVC**
BuildingRESTAPIsand web controllers

**Spring Data JPA**
Databaseoperationswithzero SQL

**AOP**
Cross-cutting concerns: logging, security

**Real Projects**
Putting everything together

All examples with full code on GitHub:
**github.com/va-run04/SpringBoot-Learning-**

**Save · Share · Follow for more**

Next up: Spring MVC + Spring Data JPA