Spring Core 6

IOC Container

Spring Framework uses an **IoC (Inversion of Control) container** to create and manage objects. The IoC container is created by reading configuration metadata, such as the **beans.xml file**. In this context, a Spring Bean refers to any object that is created and managed by the IoC container within the Spring Framework.

There are two primary ways to use the Spring IOC container:

- BeanFactory (Interface)
- ApplicationContext (Interface)

Among these, ApplicationContext offers a richer set of features compared to BeanFactory, making it the preferred choice. ApplicationContext follows the Factory Design Pattern, and one of its common implementations is ClassPathXmlApplicationContext.

Dependency Injection

Dependency Injection (DI) refers to the process of supplying the necessary dependencies to a class.

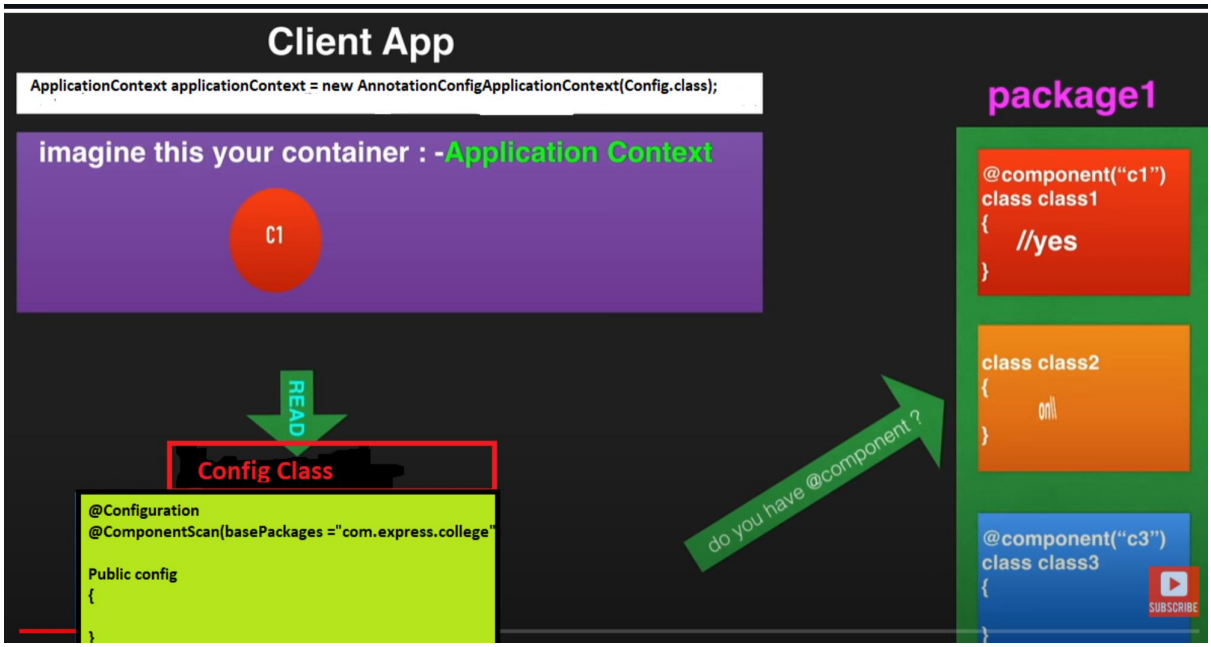
There are two main types of dependency injection:

- Constructor Injection
- Setter Injection
- Field Injection

Autowired Annotation

The @Autowired annotation in Spring is used to **automatically inject dependencies into a class**, eliminating the need for explicit bean writing in configuration files. It supports injection on fields, setter methods, and constructors, enabling Spring to resolve and inject the required beans by type.

Activating Dependency Injection Using Spring Annotation



Add Spring Dependencies to Your Project

If using Java-based configuration, annotate a configuration class with @Configuration and enable component scanning with

@ComponentScan:

```
@Configuration
@ComponentScan                    "com.example.package"
public class AppConfig
```

@Configuration marks a class as a configuration class

Spring reads the configuration class to create ApplicationContext.

When the application starts, Spring scans for classes annotated with @Configuration. This process results in the creation of the ApplicationContext, which holds and manages these beans throughout the application lifecycle.

Constructor Injection

Constructor injection in Spring is a dependency injection technique where dependencies are provided to a class through its constructor.

- When the context is initialized, Spring scans for components.
- It finds the Engine and Car beans.
- For Car, Spring sees the constructor requiring an Engine.
- Spring looks up the Engine bean and injects it into the Car constructor.
- Car instance is created with the injected Engine.

- The Car bean is now ready to be used with all dependencies satisfied.

Note : Since Spring 4.3+, if a class has only one constructor, **@Autowired is optional.**

Setter Injection

Setter Injection in Spring using annotations is a way to inject dependencies into a Spring-managed bean by calling setter methods annotated with @Autowired.

- Spring container starts by scanning the classpath for components (e.g., classes annotated with @Component).
- When it finds a bean class (e.g., MessageSender), it creates an instance of that bean by calling the no-argument constructor.
- It looks for setter methods annotated with @Autowired.
- Spring calls the setter method on the bean instance, passing the resolved dependency as an argument.
- This injects the dependency into the bean.

Field Injection

Field injection in Spring using the @Autowired annotation works by Spring's container performing dependency injection directly into the fields of a bean after the bean instance has been created.

- Spring scans the specified package (com.springcore.ioc) and finds all classes annotated with @Component.
- Spring Finds Keyboard and Computer Classes.

- For each detected component, Spring creates a bean by calling the default constructor.

- After creating the Computer instance, Spring looks at its fields.

- It finds the keyboard field annotated with @Autowired.

- Spring looks up the bean of type Keyboard in the application context.

- Using Java reflection, Spring sets the keyboard field of the Computer instance to the Keyboard bean.

Why is Constructor Injection Prefered Over Field Injection ?

- **Immutability and final Fields :** Constructor injection allows you to declare dependencies as final, making your classes immutable after construction. Field injection happens after the constructor runs, so it cannot inject into final fields. Immutability improves thread safety and design clarity.

- **Explicit Dependencies and Better Design :** With constructor injection, all required dependencies are explicit in the constructor signature. This makes it clear what a class needs to function, supporting the Single Responsibility Principle (SRP) and making the code easier to understand and

maintain. Field injection hides dependencies inside the class, reducing clarity.

- **Easier and More Reliable Unit Testing :** Constructor injection enables you to create instances of your classes manually in unit tests by passing mock dependencies directly through the constructor. Field injection requires the Spring container or reflection hacks to inject mocks, complicating testing.

- **No Reflection Needed for Injection :** Constructor injection does not rely on reflection to set private fields; dependencies are passed explicitly. This improves performance and avoids breaking encapsulation.

When Field Injection Can Be Used

- When you have optional dependencies that are not critical for the object's construction (though setter injection is usually better for optional dependencies).

- If you are working in legacy code or quick prototypes where refactoring to constructor injection is not feasible.

Spring Core Annotations

Bean Annotation

The @Bean annotation in Spring is a method-level annotation used to indicate that a method produces a bean to be managed by the Spring IoC (Inversion of Control) container.

Declaration: You declare a bean by annotating a method with @Bean inside a class annotated with @Configuration or @Component.

@Component vs @Bean Annotation

Aspect	@Component	@Bean
Annotation Level	Class-level	Method-level
Registration Style	Automatic via classpath scanning	Explicit declaration in @Configuration class
Use Case	Own application classes	Third-party or external classes, complex bean creation
Bean Naming	Defaults to class name (can be customized with @Component("name"))	Defaults to method name (can be customized)

Example :

```
// Using @Component for your own class
@Component
public class MyService
    // Spring auto-registers this bean


// Using @Bean to register a third-party class
@Configuration
public class AppConfig
```

In this example, MyService is automatically registered because it is annotated with @Component. The ObjectMapper bean is explicitly declared via @Bean because it is a third-party class you cannot annotate directly

@Primary Annotation

The @Primary annotation is used to indicate which bean should be given preference when multiple beans of the same type are candidates for autowiring.

@Primary is placed on the bean definition, which means it can be applied:

- At the method level on a @Bean method inside a @Configuration class.

Example :

@Qualifier Annotation

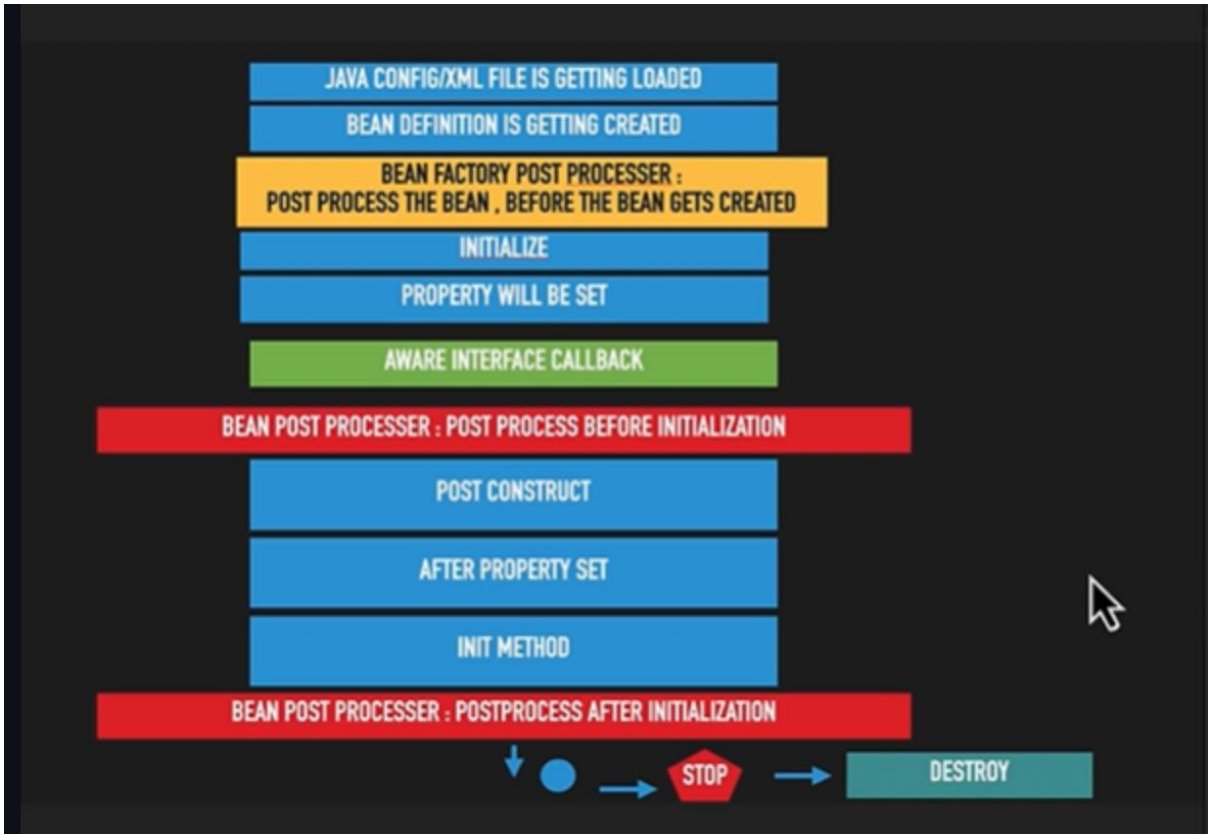
@Qualifier is used at the injection point to explicitly specify which bean to inject when multiple beans of the same type exist. It overrides the @Primary preference.

@Qualifier is used at the injection point, typically:

- On a field, constructor parameter, or setter method where the bean is injected.

Bean Life Cycle

The Spring Bean life cycle is the process that a Spring-managed bean goes through from its creation to its destruction within the Spring container.



- **Bean Definition Loading** :

	Action	Description
1	Create ApplicationContext	`new AnnotationConfigApplicationContext()` creates an empty context
2	Register Configuration Class	`context.register(AppConfig.class)` adds a config class

3	Parse `@Configuration`, find `@Bean` methods	These methods are registered as bean definitions (but not yet run)
4	**Perform Component Scanning** (if enabled)	Looks for `@Component`, `@Service`, etc., and registers them as bean definitions
5	Result:	At this point, Spring knows *what* beans to create — but has *not* created them yet

- **BeanFactoryPostProcessor :** Before any bean instances are created, Spring calls all registered BeanFactoryPostProcessor implementations.

    1. These processors can modify bean definitions (metadata), such as changing property values, scopes, or adding new bean definitions.

    2. This step happens after bean definitions are loaded but before any beans are instantiated.

- **Bean Instantiation :** The container creates an instance of the bean by calling its constructor. If the bean has a no-argument constructor, it is called by default.

- **Dependency Injection :** Spring inject dependencies into the bean's properties via setters, constructor, or autowiring.

- **Aware Interfaces Callback :** If implemented, Spring calls methods like setBeanName(), setBeanFactory(), and

setApplicationContext().

- **BeanPostProcessor Before Initialization :** Spring calls postProcessBeforeInitialization() methods of any registered BeanPostProcessors, allowing modification of the raw bean instance before initialization callbacks.

- **@PostConstruct Method Execution :** Spring invokes methods annotated with @PostConstruct. This happens after dependency injection and before other init callbacks.

- **InitializingBean.afterPropertiesSet()**
  If implemented, Spring calls the afterPropertiesSet() method.

- **Custom init-method**
  If configured, Spring calls the custom init method specified in XML or Java config.

- **BeanPostProcessor After Initialization :**
  Spring calls postProcessAfterInitialization() methods of any registered BeanPostProcessors, allowing for proxy wrapping or further processing.

- Bean Ready for Use
  The bean is now fully initialized and ready for use.

- Application Context Shutdown / Bean Destruction

- **@PreDestroy Method Execution**
  Spring calls methods annotated with @PreDestroy before other destruction callbacks.

- DisposableBean.destroy()
  If implemented, Spring calls the destroy() method.

- Custom destroy-method
  If configured, Spring calls the custom destroy method.

BeanFactoryPostProcessor vs BeanPostProcessor

Aspect	**BeanFactoryPostProcessor**	**BeanPostProcessor**
Execution Time	After loading bean definitions, before instantiation	After bean instantiation and initialization
Operates On	Bean definitions (metadata/configuration)	Bean instances (actual objects)
Lifecycle Phase	Pre-instantiation	Post-instantiation
Example Interface Method	postProcessBeanFactory(ConfigurableListableBeanFactory)	postProcessBeforeInitialization(), postProcessAfterInitialization()

Bean Scope

Spring manages bean instances according to their scope, which defines the lifecycle and visibility of a bean within the container.

Singleton Bean

A singleton bean in Spring means that the Spring IoC container creates exactly one instance of the bean per container.

- **Default Scope:** Singleton is the default scope in Spring. If you do not specify any scope, your bean is singleton by default.

- **Per Container:** Unlike the classic singleton design pattern (one instance per JVM/classloader), Spring's singleton scope is per Spring container. Multiple containers can have separate singleton instances.

```
@Bean
public                    mySingletonBean
    return new
```

OR

Whenever we annotate a class with @Component its by default scope in Singleton and it will be eagerly initialised.

```
                                                    // Outputs: true
```

Prototype Bean

The prototype scope in Spring means that a new bean instance is created every time the bean is requested from the Spring container.

- **New instance per request:** Each call to getBean() or injection point results in a new bean instance.

- **Usage:** Use prototype scope when you want independent bean instances, such as user sessions, form backing objects, or any stateful component.

```
@Component
@Scope                                                    //
Marks this bean as prototype scoped
public class MyPrototypeBean

    public MyPrototypeBean
                                "Prototype bean instance
created: "   this

    public void showMessage
                                "Hello from prototype
bean: "   this

```

```
public class MainApp

    public static void main

new

                                "Requesting prototype bean
first time:"
```

```
                                                    "Requesting prototype bean
second time:"
                                                    "Are both instances the
same? "
```

Note : When you have a singleton-scoped bean that depends on a prototype-scoped bean in Spring, by default, the prototype bean is instantiated only once at the time the singleton bean is created and injected into it.

This means :

- The singleton bean holds a single instance of the prototype bean for its entire lifecycle.
- Even though the prototype scope means "new instance every time," this behavior is lost when injected directly into a singleton because dependency injection happens only once during singleton instantiation.

How to Get a New Prototype Instance Every Time in a Singleton?

Using @Lookup Method Injection

```
@Component
@Scope "prototype"
public class Student
    public Student
                                "New Student instance
created: "   this
    public void study
                                "Student "   this   " is
studying."
```

```
@Component
public class SchoolService

    public void processStudent
                                                // Spring
overrides this method to return a new prototype bean

    @Lookup
    protected          getStudent
        // This method is overridden by Spring to
return a new Student prototype instance on each call
        return null
```

```

```

```
public class MainApp
    public static void main
new

// Creates and
uses first prototype Student
// Creates and
uses second prototype Student

```

Output : Here 2 new Students will be created

Spring uses CGLIB to generate a subclass of the singleton bean where the @Lookup method is overridden.

Spring Proxy

A proxy in Spring is an object created dynamically at runtime that wraps the real target object.

Why Does Spring Use Proxies?

- To implement Aspect-Oriented Programming (AOP): Proxies allow Spring to apply aspects like logging, security, or transaction management around method executions without changing the business logic code.

- To enable transaction management: Spring wraps service objects in proxies that start, commit, or rollback transactions automatically when methods are called.

- To implement other cross-cutting concerns like caching, auditing, and performance monitoring.

Spring uses two main proxying mechanisms :

- JDK Dynamic Proxies
- CGLIB Proxies

JDK Proxy

JDK Dynamic Proxy is used when :

- The target object implements at least one interface.
- Spring uses the java.lang.reflect.Proxy class to create a proxy.

How JDK Dynamic Proxy Works :

- Define Interface

```
public interface PaymentService
    void pay double
```

- Provide Real Implementation

```
public class CreditCardPaymentService implements
PaymentService
    @Override
    public void pay double
                              "Paying "              " using
Credit Card."
```

- Create Invocation Handler

```
public class LoggingInvocationHandler implements
InvocationHandler
    private final

    public LoggingInvocationHandler
        this

    @Override
    public        invoke
                  throws
                              "[LOG] Before method: "
                                                                //
call real method
                              "[LOG] After method: "
        return
```

- Create Proxy and Use

```
                                    new
                                    
                                    
                                    
                                    
                                    
// Class loader
                new
// Interfaces to implement
                new
// Method interceptor
                
                
                  1000
```

Internally :

- Java intercepts the call to pay().
- The invoke() method of LoggingInvocationHandler is called.
- Your custom logic runs (e.g., logging, security).
- The real method (realService.pay()) is invoked.
- The result is returned.

Disadvantage :

- JDK dynamic proxies can only proxy interfaces, so the target class must implement at least one interface.

- Method invocations on JDK dynamic proxies use Java reflection, which is slower than direct method calls.

Note : For Program please refer **com.springcore.jdkproxy**

Why Use JDK Dynamic Proxy in Spring?

- To implement AOP features like transactions, security, and logging without modifying business code.
- To keep cross-cutting concerns separate and modular.

CGLIB Proxy

CGLIB (Code Generation Library) proxy is a proxy mechanism used by Spring to create runtime-generated subclasses of target classes.

How CGLIB(Code Generation Library) Proxy Works :

Example - 1: Using ProxyFactoryBean

```
                    new
        //creating proxy factory
                                        new
                    // getting object from factory
                    // proxy object obtained using CGLIB
```

Example - 2 : Using Enhancer

```
    final                        new
                                new

    // for target class

                            new

// which ever method name //MethodProxy has more
functionalites then Method
        public         intercept
                                            throws

                                "before invoking method"
        // method = getSize()
// which inturn calls actual method of RAM Class.


                                "after invoking method"
                       return


// creating CGLib Proxy for target class already defined



                                "Proxy class name: "

        // which will call anonymous method called -->
intercept()

```

How Can We Assure that Spring is creating a CGLIB proxy ?

Make the RAM class as **Final** then CGLIB will fail because CGLIB works by **creating a subclass (proxy)** of the target class.

Will get Runtime Exception:

```
final class com example RAM
```

Question : You have a singleton bean (only one instance created by the Spring container).
Inside it, you @Autowired a prototype bean (which is expected to return a new instance on each request).
However, you don't get a new instance every time.
Why does this happen?

Spring injects the prototype bean only once — at the time the singleton is created.

- Even though the prototype bean is defined as prototype,
- The singleton bean holds the same instance throughout its lifecycle.

Note : For Program please refer xom.springcore.prototypescope
Without adding **proxyMode** into **RAM class**

How to Fix It (Get New Prototype Every Time)

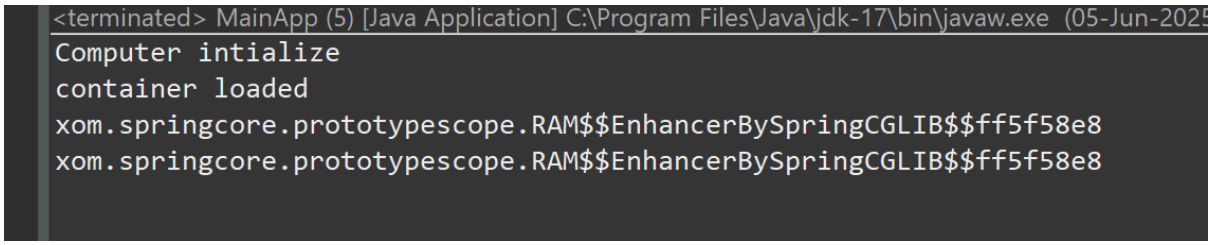
By Using Annotation on top of RAM Class :

@Scope(value = "prototype", proxyMode
ScopedProxyMode.TARGET_CLASS)

What Happens behind Scenes ?

- Normally, Spring injects the actual prototype instance at startup. But when proxyMode is used, Spring injects a proxy object into the singleton.
- Instead of injecting the real RAM, Spring injects a proxy.
- That proxy extends RAM (because of TARGET_CLASS) and overrides its methods.
- When I execute RAM ram1 = computer.getRam(); I receive a proxy object.
- Therefore, when I print it using System.out.println(ram1.getClass().getName()); it displays the new proxy RAM object.

Output :



Disadvantages :

- CGLIB proxies generally have higher creation and invocation overhead compared to JDK dynamic proxies because they involve generating subclasses and bytecode manipulation at runtime.

- Cannot Proxy final Classes or Methods.

- CGLIB requires the target class to have a default (no-argument) constructor by default.