Java Spring Framework

Starting with Spring Framework (Without Spring Boot)

Why Move from Spring Boot to Spring Framework

- Spring Boot makes development easy using:
  - @Component
  - @Autowired
- However:
  - It hides what happens **behind the scenes**
- To understand **core Spring internals**, we must:
  - Write code **without Spring Boot**
  - Work directly with **Spring Framework**

Creating a Pure Spring (Core) Project

- IDE used: **IntelliJ IDEA** (Eclipse also works)
- Project type: **Maven Project**
- Reason:
  - Maven ensures the **same project structure** across IDEs
- Steps:
  - File → New Project
  - Choose **Maven**
  - Select **Quick Start archetype**
  - Project name: spring-one
  - Group ID: com.telusko
  - Artifact ID: springone
  - Java version:
    - **Minimum JDK 17** (required for Spring 6)
    - Java 17 / 18 / 19 / 20 / 21 all work

Verifying Maven Project Setup

- Maven creates a default structure
- Default class prints:
  - "Hello World!"

- Best practice:
  - Run the project **before making changes**
  - Confirms environment and setup are correct

Creating a Simple Alien Class

- New class: **Alien**
- Added method:
  - code() → prints "Started Coding.."
- Initially:
  - Object created using new Alien()
  - Method called directly
- This confirms:
  - Basic Java setup works correctly

Goal: Let Spring Create the Object

- Requirement:
  - Avoid using new
  - Let **Spring manage object creation**
- To do this:
  - A **Spring IoC Container** must be created
- The container is accessed via:
  - **ApplicationContext**

Understanding ApplicationContext

- ApplicationContext:
  - Interface provided by Spring
  - Manages the **IoC container**
- Alternatives:
  - **BeanFactory** (older, mostly deprecated)
- Key point:
  - ApplicationContext is a **superset** of BeanFactory
  - Provides additional enterprise features
- Spring 6:
  - Many BeanFactory implementations are deprecated or removed
  - ApplicationContext is recommended

Adding Spring Dependency (spring-context)

- ApplicationContext is **not part of Java**
- Must add Spring dependency manually
- Steps:
  - Open pom.xml
  - Add **spring-context** dependency
  - Version used:
    - Spring **6.1.x**
    - Recommended: **second latest version**
- After adding dependency:
  - Reload Maven
  - Spring libraries appear under **External Libraries**

———

Choosing XML-Based Configuration

- Spring supports multiple configuration styles:
  - XML-based
  - Java-based
  - Annotation-based
- In this section:
  - **XML-based configuration** is used
- Chosen implementation:
  - ClassPathXmlApplicationContext

———

Creating the IoC Container

- Code used:
  - ApplicationContext context = new ClassPathXmlApplicationContext();
- Purpose:
  - Creates a **Spring container**
  - Allows interaction with managed beans

———

Fetching a Bean from the Container

- Method used:
  - context.getBean("alien")
- Behavior:
  - Returns an Object
- Required step:

        - Typecasting to Alien
    - Code:
        - (Alien) context.getBean("alien")

Runtime Error Encountered

- Error message:
    - BeanFactory not initialized or already closed
    - Suggests calling refresh()
- Cause:
    - Container was created **without configuration**
    - No XML file was provided
    - Spring does not know:
        - Which beans to create
        - How to initialize the container

Key Takeaway

- Unlike Spring Boot:
    - Spring Framework requires **explicit configuration**
- Important concepts introduced:
    - IoC Container
    - ApplicationContext
    - XML-based configuration
- Next step:
    - Define Spring configuration properly
    - Understand **why refresh() is required**
    - Learn how Spring initializes beans internally

```
package org.springframeworkxmlbasedconfig;

public class Alien {
    public void code() {
        System.out.println("Started Coding.. ");
    }
}
```

```
package org.springframeworkxmlbasedconfig;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCont
ext;

public class Main {
    public static void main(String[] args) {

        // if you want to create a container, we have to use
something called an ApplicationContext.Now ApplicationContext
is
        // responsible to work with the IOC container. Actually
we do have two options to interact with the container
        // One is BeanFactory which is an old one, the
application context, a new one. Both does the same thing,
almost the same thing.
        // Basically, it will help you to create the container,
and it will help you to get the objects from the container


        ApplicationContext context = new
ClassPathXmlApplicationContext();
        // Now if you go to application context it's an it's
basically an interface.It extends a lot of different or
interfaces.
        // and then if I click on this particular
ListableBeanFactory it extends Bean Factory. So by default, or
we can say this
        // application context is a superset of Bean Factory.So
whatever we want to do with Bean Factory can be done with
application context.
        // But the question arises how will you create the
object.What are the classes which implements this.Actually
there are a lot of classes which implements this.
        // We are going to use the
ClassPathXmlApplicationContext.
        // See, the thing is there are different ways of
configuring your spring project and one of them is XML. So in
this particular section we are going to talk about XML.
        // Then we'll move towards Java based and then we'll
move towards the annotations.
        // ApplicationContext context = new
ClassPathXmlApplicationContext("alien"); - which actually
```

```
creates the container and then with that container
        // we can specify what class object we want
        Alien alien = (Alien) context.getBean("alien");
        // But the thing is by default, getBean will give you
the type of object as object. If you can see it says it is
providing object.
        // Object getBean(String name) throws BeansException;
But what we want is alien object. So what you have to do is we
have to just do typecasting.
        // Alien alien = new Alien();
        alien.code();
    }
}
```

Object Creation in Spring Using XML Configuration

Overview of Spring-Managed Object Creation

- Spring Framework is responsible for creating objects (beans) and managing them inside the **Spring Container**.
- Developers do **not** create objects using new; instead, Spring creates them and provides references using getBean().
- Once the object reference is obtained, its methods can be invoked normally.

———

Understanding When Spring Creates Objects

Key Observation

- Objects are created **when the Spring container is initialized**, not when getBean() is called.

Critical Line of Execution

ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");

- This line:
  - Creates the **Spring IOC container**
  - Loads the XML configuration file

- Instantiates **all beans defined with <bean> tags**

Using Constructors to Verify Object Creation Timing

Alien Class Constructor

```
public Alien() {
    System.out.println("Object Created");
}
```

Result

- The constructor message prints **even if getBean() is commented out**
- Confirms that object creation happens during container initialization

---

Role of getBean() in Spring

What getBean() Does

- Returns a **reference** to an already-created object in the container
- Does **not** create a new object

Example

Alien alien = (Alien) context.getBean("alien1");

- Retrieves the bean named alien1 from the container
- Typecasting is required because getBean() returns Object

Bean Creation Depends on <bean> Definitions

XML Configuration Example

```
<bean id="alien1"
class="org.springframeworkxmlbasedconfig.Alien"/>
<bean id="lap" class="org.springframeworkxmlbasedconfig.Laptop"/>
```

Behavior

- Spring creates:
  - One Alien object

        - One Laptop object
    - Each <bean> tag results in **one object creation**

Multiple Beans of the Same Class

Multiple Bean Definitions

```
<bean id="alien1"
class="org.springframeworkxmlbasedconfig.Alien"/>
<bean id="alien2" class="org.springframeworkxmlbasedconfig.Alien"/>
```

Result

- Spring creates **two separate objects**
- Each bean ID corresponds to a distinct instance
- Constructors are invoked once per bean definition

Effect of Bean Count on Object Creation

Summary

- Number of objects created = number of <bean> tags
- Same class + different IDs → multiple objects
- Different classes → separate objects

Bean Without an ID

Important Note

- Spring **can** create a bean even if id is not specified
- However:
    - Without an ID, the bean **cannot be referenced** using getBean(String name)
- ID is required for retrieval, not creation

Using Multiple References to the Same Bean

Example

```
Alien obj1 = (Alien) context.getBean("alien1");
```
Alien obj2 = (Alien) context.getBean("alien1");

Discussion Point Raised

- Question posed:
  - Will Spring create one object or two?
- Clarification deferred to the next section/video
- Based on prior discussion:
  - Object creation depends on <bean> count, not getBean() calls

---

Spring Container and Configuration Mechanism

ApplicationContext

- ApplicationContext is used to interact with the IOC container
- It is a **superset of BeanFactory**
- Provides additional enterprise-level features

Implementation Used

```
ClassPathXmlApplicationContext
```

- Loads XML configuration from the classpath
- Reads <bean> definitions and initializes objects

Laptop Class Example

Laptop Constructor

```
public Laptop() {
    System.out.println("Laptop Object Created.. ");
}
```

Observation

- Constructor is executed only if a corresponding <bean> tag exists
- Confirms Spring does not scan or create objects automatically

---

Key Takeaways

- Spring creates objects **during container initialization**
- <bean> tags determine:
  - Which classes are instantiated
  - How many objects are created
- getBean():
  - Retrieves existing objects
  - Does not trigger object creation
- Multiple bean definitions → multiple objects
- ID is optional for creation but required for retrieval

Below are **structured, transcript-faithful notes** covering the continuation of the Spring XML configuration topic, written clearly for sharing and revision.
Only information present in your provided transcript and code is used.

When Exactly Does Spring Create an Object?

Initial Observation

- Spring is responsible for creating objects (beans).
- getBean() is used only to **retrieve** the object reference.
- Methods are called using that reference, not during creation.

Identifying the Exact Line Where Object Is Created

Two Possible Lines

- Line where ApplicationContext is created
- Line where getBean() is called

Experiment Using Constructor

- A constructor is added to the Alien class:

```
public Alien() {
    System.out.println("Alien Object Created...");
}
```

Test Performed

- getBean() and method calls are commented out
- Only this line remains:

```
ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
```

Result

- "Alien Object Created..." is printed
- Confirms that:
  - Object is created **when the container loads**
  - Not when getBean() is called

What Happens During ApplicationContext Initialization

Key Behavior

- Spring:
  - Creates the IOC container
  - Reads the XML configuration file
  - Instantiates all classes defined using <bean> tags

Important Conclusion

- getBean() does **not** create objects
- It only fetches an existing object from the container

Bean Creation Depends on XML Configuration

Single Bean Scenario

- Two classes exist: Alien and Laptop
- XML contains only:

```
<bean id="alien" class="Alien"/>
```

Result

- Only Alien object is created
- Laptop constructor is not called

Creating Multiple Beans for Different Classes

Updated XML

```
<bean id="alien" class="Alien"/>
<bean id="lap" class="Laptop"/>
```

Result

- Two objects created:
  - One Alien
  - One Laptop

Conclusion

- Spring creates objects **only for classes mentioned as beans**
- Number of objects = number of <bean> tags

Multiple Beans of the Same Class

XML Configuration

```
<bean id="alien1" class="Alien"/>
<bean id="alien2" class="Alien"/>
```

Result

- Alien constructor runs **twice**
- Spring creates **two separate objects**

Key Point

- Same class + different bean IDs = multiple objects

Bean ID Is Optional for Creation

Observation

- Bean can be created even without an id
- However:
  - Without an ID, it cannot be referenced using getBean()

Multiple References to the Same Bean

Code Example

```
Alien alien1 = (Alien) context.getBean("alien1");
Alien alien2 = (Alien) context.getBean("alien1");
```

Question Raised

- Will Spring create one object or two?

Proof Using Instance Variable

Alien Class

```
int age;
```

Test

```
alien1.age = 28;
System.out.println(alien1.age); // 28
System.out.println(alien2.age); // 28
```

Result

- Both references print 28
- Confirms:
  - Both references point to the **same object**

Default Bean Scope: Singleton

Explanation

- By default, Spring beans use **singleton scope**
- Meaning:
  - Only **one object per bean definition**
  - Same object is returned for every getBean() call

Key Behavior

- Calling getBean() multiple times does **not** create new objects
- All references point to the same instance

Introduction to Bean Scopes

Scopes Mentioned

- singleton
- prototype
- request
- session

Spring Core Focus

- Only two commonly used:
  - singleton
  - prototype
- request and session are used in web applications

Effect of Prototype Scope

XML Example

```
<bean id="alien1" class="Alien" scope="prototype"/>
```

Observed Behavior

- When only this line runs:

```
ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
```

Output

- Only:

```
Laptop Object Created..
```

Reason

- prototype beans:
  - Are **not created at container load time**
  - Are created **only when getBean() is called**

Singleton vs Prototype Object Creation Timing

Singleton

- Object is created:

        - When the container loads
    - Default Spring behavior

Prototype

- Object is created:
    - Only when getBean() is called

Final Recap

- Spring creates objects when the container loads **if scope is singleton**
- Number of objects depends on:
    - Number of <bean> tags
    - Bean scope
- getBean():
    - Returns an existing object
    - Does not create objects in singleton scope
- Multiple references can point to the same object
- Prototype scope changes creation timing behavior

Introduction to Constructor Injection

- The Alien class contains **properties**:
    - A primitive variable: age
    - A reference variable: Laptop
- These properties must be **initialized**:
    - Primitive → assign a value
    - Reference → inject an object
- Earlier, this was done using **setter injection** via <property> tags in XML.
- In Java, initialization can also be done using **constructors**, which is often preferred when values are required at object creation time.

Setter Injection vs Constructor Injection

Setter Injection

- Uses setter methods like setAge() and setLaptop()
- Values are assigned **after** object creation
- Implemented using <property> tag in XML

Constructor Injection

- Uses a **parameterized constructor**
- Values are assigned **at the time of object creation**
- Implemented using <constructor-arg> tag in XML

Using a Parameterized Constructor

Alien Constructor

```
public Alien(int age) {
    this.age = age;
}
```

- This constructor assigns age when the object is created.
- To verify constructor usage, a log statement is added:

```
System.out.println("Parameterized Constructor called...");
```

Using <constructor-arg> in XML

Basic Constructor Injection

```
<bean id="alien1"
class="org.springframeworkxmlbasedconfig.Alien">
    <constructor-arg value="21"/>
</bean>
```

Result

- The parameterized constructor is called
- Default constructor is not invoked
- Confirms constructor-based value injection

Constructor Injection with Multiple Parameters

Updated Constructor

```
public Alien(int age, Laptop laptop) {
    this.age = age;
    this.laptop = laptop;
}
```

- Now the constructor expects:
    - One primitive (int)
    - One object reference (Laptop)

Error When Arguments Are Missing

Problem

- XML provides only one constructor argument
- Constructor expects two

Error

- Unsatisfied dependency
- Spring cannot resolve the missing Laptop dependency

Injecting Object References via Constructor

Correct XML Configuration

```
<bean id="alien1"
class="org.springframeworkxmlbasedconfig.Alien">
    <constructor-arg value="21"/>
    <constructor-arg ref="lap"/>
</bean>
```

- value → primitive (age)
- ref → object reference (Laptop bean)

How Spring Matches Constructor Arguments

Initial Assumption

- Spring matches arguments based on **type**

Reality
- By default, Spring matches constructor arguments by **sequence (order)**

Error Example
- Swapping arguments causes:

```
cannot convert argument value of type Laptop to int
```

Solution 1: Using type Attribute

XML Example

```
<constructor-arg type="int" value="28"/>
<constructor-arg type="org.springframeworkxmlbasedconfig.Laptop" ref="lap"/>
```

Behavior
- Spring matches arguments by **type**, not order

Limitation
- Works only if constructor parameters are of **different types**
- Fails when multiple parameters have the same type (e.g., two ints)

Problem with Same Data Types

Scenario

```
private int age;
private int salary;
```

- Constructor accepts two int parameters
- Spring cannot decide which value belongs to which variable

Solution 2: Using index Attribute (Recommended)

XML Example

```
<constructor-arg index="0" value="28"/>
<constructor-arg index="1" ref="lap"/>
```

Key Points

- Index starts from **0**
- Index corresponds to constructor parameter order
- Works even when:
    - Multiple parameters have the same type
- Most reliable and commonly used approach

Solution 3: Using name Attribute

XML Example

```
<constructor-arg name="laptop" ref="lap"/>
<constructor-arg name="age" value="28"/>
```

Important Note

- By default, **sequence must still match**
- If order is changed, it will fail unless additional configuration is used

---

Using @ConstructorProperties Annotation

Purpose

- Allows Spring to map constructor arguments by **parameter name**
- Removes dependency on sequence order

Alien Constructor

```
@ConstructorProperties({"age", "laptop"})
public Alien(int age, Laptop laptop) {
    this.age = age;
    this.laptop = laptop;
}
```

Result

- XML can now use name safely
- Constructor injection works even if order changes

Constructor Injection Summary

Ways to Resolve Constructor Arguments

1. **By sequence (default behavior)**
2. **By type**
    - Only works with different data types
3. **By index** ✅ (most reliable)
4. **By name**
    - Requires @ConstructorProperties

When to Use Constructor Injection vs Setter Injection

Constructor Injection

- Use when properties are **mandatory**
- Ensures object is always in a valid state

Setter Injection

- Use when properties are **optional**
- Allows modification after object creation

Final Recommendation

- Prefer **constructor injection** for required dependencies
- Use **index-based constructor arguments** for clarity and reliability
- Use setter injection only for optional properties

Problem with Tight Coupling (Alien Depends on Laptop)

- Initially:
    - Alien class directly depends on Laptop
- Issue:
    - Alien is tightly coupled to a specific implementation

        - Reduces flexibility
    - Real-world analogy:
        - To code, you need a **computer**, not specifically a **laptop**
        - A computer can be:
            - Laptop
            - Desktop
            - Any future device

Introducing an Interface to Reduce Dependency

- A new interface **Computer** is created
- Purpose:
    - Act as an abstraction layer
- Interface characteristics:
    - Represents a concept
    - Does not have direct object instantiation
- Contains:
    - A single method: compile()

Laptop Implementing Computer Interface

- Laptop class implements Computer
- Provides its own implementation of compile()
- Example behavior:
    - Prints: *Compiling using Laptop*

Desktop as Another Implementation of Computer

- A new class Desktop is created
- Desktop also implements Computer
- Implements compile() method
- Example behavior:
    - Prints: *Compiling using Desktop*

Alien Class Depends on Interface, Not Implementation

- Alien now depends on **Computer**, not Laptop
- Benefits:
    - Loose coupling
    - Flexibility

        - Future-proof design
    - Alien does not care about:
        - Whether it is a Laptop or Desktop
    - Behavior depends on:
        - The object injected at runtime

Polymorphism Through Interface Implementation

- Computer interface has multiple implementations:
    - Laptop
    - Desktop
- Runtime behavior:
    - If Laptop is passed → Laptop's compile() is called
    - If Desktop is passed → Desktop's compile() is called
- Alien class remains unchanged regardless of implementation

Alien Class with Dependency Injection Setup

```
private int age;
private Computer comp;

public Alien() {
    System.out.println("Alien Object Created...");
}

@ConstructorProperties({"age", "computer"})
public Alien(int age, Computer comp) {
    this.age = age;
    this.comp = comp;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    System.out.println("Setter Called...");
    this.age = age;
}

public Computer getComputer() {
```

```
    return comp;
}

public void setComputer(Computer comp) {
    this.comp = comp;
}

public void code() {
    System.out.println("Started Coding.. ");
    comp.compile();
}
```

Key Outcome of This Design

- Alien:
  - Is no longer dependent on a concrete class
  - Depends on an abstraction (**Computer**)
- New implementations can be added without modifying Alien
- Demonstrates:
  - Interface-based design
  - Loose coupling
  - Foundation for **Spring Dependency Injection**

Purpose of This Setup

- This interface-based structure prepares for:
  - **Autowiring**
- Autowiring will be explained in the next section
- This design choice will make dependency injection clearer and more effective

Running the Code After Interface Creation

- At this stage:
  - One **interface** (Computer)

        - Two **implementations** (Laptop, Desktop)
    - Code was not executed in the previous step
    - When the application is run:
        - An **error occurs**
        - Error message indicates:
            - *Cannot resolve matching constructor*

Constructor-Based Injection Issue

- The error occurs because:
    - The configuration is still dependent on **constructor injection**
    - Spring cannot find a matching constructor
- To fix this:
    - Constructor-based injection is commented out
    - The configuration is switched back to **property-based injection**

Switching to Property-Based Injection

- Property injection is used for:
    - age
    - computer
- Example:
    - age is set using a <property> tag
- At this point:
    - Alien was earlier dependent on Laptop
    - Now Alien depends on Computer

Invalid Property Error Explained

- Even after switching to property injection:
    - Application still fails
- Error message:
    - *Invalid property 'lab'*
- Reason:
    - Alien no longer has a property named lab
    - Alien now has:

private Computer comp;

- Fix:

- Update XML to refer to comp instead of lab

Manual Wiring Using Property Reference

- The comp property is explicitly wired using a reference:

```
<property name="comp" ref="lab1" />
```

- lab1 refers to a Laptop bean
- When run:
  - Output shows:
    - *Compiling using Laptop*
- Reason:
  - Laptop object is explicitly injected

Cleaning Up the Code Output

- Unnecessary print statements are removed:
  - Object creation logs
  - Setter logs
- Final clean output:
  - *Started Coding*
  - *Compiling using Laptop*

Using the Same Name for Bean and Property

- Bean ID and property name are both changed to comp
- Example:

```
<bean id="comp" class="Laptop"/>
```

- Result:
  - Application works correctly
- Conclusion:
  - Bean name and property name can be the same without issues

---

What Happens If Property Is Not Defined

- The comp property is removed from configuration
- Result:
  - Runtime error occurs
- Error reason:
  - comp is null
  - Spring does not inject anything automatically by default

Need for Automatic Wiring

- Manual property wiring works but is repetitive
- Question raised:
  - Can Spring automatically link dependencies?
- Answer:
  - Yes, using **Autowiring**

Autowiring by Name

- Configuration:

```
<bean id="alien1" class="Alien" autowire="byName">
    <property name="age" value="28"/>
</bean>
```

- Behavior:
  - Spring looks for a bean whose **ID matches the property name**
  - Property name: comp
  - Bean ID: comp
- Result:
  - Dependency injected automatically
  - Output:
    - *Compiling using Laptop*

Multiple Implementations with Autowire by Name

- Two beans defined:
  - comp → Laptop
  - comp1 → Desktop
- Spring selects:
  - The bean whose ID matches comp
- Result:

- Laptop is used

Switching Autowire Target Using Name Change

- Bean ID is changed to match Desktop
- Spring behavior:
  - Selects Desktop automatically
- Output:
  - *Compiling using Desktop*

Explicit Property Overrides Autowiring

- Even when autowiring is enabled:
  - Explicit <property> takes priority
- Example:

<property name="comp" ref="comp1"/>

- Result:
  - Spring uses the explicitly provided reference
  - Autowiring is ignored

Autowire Works Only When Property Is Not Explicitly Set

- Rule:
  - If <property> is defined → Spring uses it
  - If <property> is missing → Spring applies autowiring

Autowiring by Name Failure Case

- Bean IDs:
  - comp1
  - comp2
- Property name:
  - comp
- Result:
  - Spring cannot find a matching bean
  - comp remains null
  - Application fails

Autowiring by Type

- Configuration:

```
<bean id="alien1" class="Alien" autowire="byType">
    <property name="age" value="28"/>
</bean>
```

- Behavior:
  - Spring ignores bean names
  - Searches by **type**
- Property type:

private Computer comp;

- Any bean implementing Computer qualifies

Autowire by Type with Single Implementation

- Only one implementation present:
  - Laptop **or** Desktop
- Result:
  - Spring injects the matching type
  - Output matches the injected implementation

Autowire by Type Failure with Multiple Implementations

- Both implementations present:
  - Laptop
  - Desktop
- Error:
  - *Expected single matching bean but found two*
- Reason:
  - Spring cannot decide between two beans of the same type

Key Takeaways from This Section

- **Autowiring** allows Spring to inject dependencies automatically
- Autowiring modes discussed:
  - **byName**
  - **byType**
- Important rules:
  - Explicit property wiring overrides autowiring
  - byName depends on matching bean IDs
  - byType fails when multiple beans of the same type exist
- This mechanism explains how:
  - @Autowired works internally in Spring Boot

Problem: Autowiring by Type with Multiple Implementations

- When using **autowire="byType"**
- And there are **multiple beans of the same type**
  - Example:
    - Laptop implements Computer
    - Desktop implements Computer
- Spring throws an error:
  - *Expected single matching bean but found two*
- Reason:
  - Spring cannot decide which implementation to inject

Real-World Analogy: Primary and Backup Resources

- Example:
  - Database connections
    - One **primary**
    - One **backup**
- When the primary fails:
  - Backup is used
- Same concept applies to Spring beans

Solution: Using primary="true"

- One bean can be marked as **primary**
- This tells Spring:

        - If there is a **conflict**, prefer this bean
    - Configuration example:

```
<bean id="comp" class="org.springframeworkxmlbasedconfig.Laptop"
primary="true"/>
<bean id="desk" class="org.springframeworkxmlbasedconfig.Desktop"/>
```

Autowiring by Type with Primary Bean

- Alien bean configuration:

```
<bean id="alien1" class="org.springframeworkxmlbasedconfig.Alien"
autowire="byType">
   <property name="age" value="28"/>
</bean>
```

- Behavior:
    - Spring detects two beans of type Computer
    - Since Laptop is marked as primary:
        - Laptop is injected
- Output:
    - *Compiling in Laptop...*

How Primary Resolves Confusion

- primary="true" is used **only when Spring is confused**
- It does **not** override explicit configuration
- It acts as a **default preference**

Explicit Property Overrides Primary

- If the dependency is explicitly specified:

```
<property name="comp" ref="desk"/>
```

- Even though Laptop is primary:
    - Spring injects Desktop

- Reason:
  - Explicit wiring has higher priority than autowiring or primary

---

Primary vs Explicit Wiring Priority

- Priority order:
  1. **Explicit property reference**
  2. **Autowiring by name**
  3. **Autowiring by type**
  4. **Primary bean (only for conflict resolution)**

Primary with Autowiring by Name

- If autowire="byName" is used:
  - Spring searches by property name
- Primary does **not** influence name-based wiring
- Bean name must still match the property name

Key Rules About Primary Beans

- primary="true":
  - Works only when multiple beans match by type
  - Used only when Spring cannot decide
- Does not apply when:
  - Dependency is explicitly wired
  - Autowiring is done by name

Final Outcome

- The multiple-bean conflict is resolved using:
  - primary="true"
- Spring behavior becomes predictable
- Dependency injection succeeds without errors

Concepts Covered in This Section

- Autowiring conflict resolution
- primary="true" usage
- Priority rules in Spring dependency injection

- Relationship between:
  - Explicit wiring
  - Autowiring
  - Primary beans

Lazy Initialization of Beans in Spring

Understanding Bean Creation in Spring

- In the application, three beans are used:
  - **Alien**
  - **Laptop**
  - **Desktop**
- By default, Spring beans have **singleton scope**
- Singleton scope behavior:
  - Only **one object per bean**
  - Object is created **when the Spring container loads**
- Container loading happens when:

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("config.xml");
```

- At this point:
  - All singleton beans are instantiated
  - Even if they are **not used**

Problem: Unused Beans Still Get Created

- Example:
  - Alien depends on Laptop
  - Desktop is **not used**
- Despite this:
  - Desktop object is still created at startup
- Proof:
  - Adding a constructor print statement:

```
public Desktop() {
   System.out.println("Desktop Object Created...");
}
```

- Output confirms object creation even when unused

Requirement: Create Bean Only When Needed

- Desired behavior:
  - Alien and Laptop should be created eagerly
  - Desktop should be created **only when requested**
- This is needed to:
  - Reduce startup time
  - Avoid unnecessary object creation

Solution: Lazy Initialization

- Spring provides a bean attribute:

```
lazy-init="true"
```

- Purpose:
  - Delays bean creation
  - Bean is created **only when first accessed**

Configuring Lazy Initialization

```
<bean id="desk"
    class="org.springframeworkxmlbasedconfig.Desktop"
    lazy-init="true"/>
```

- Result:
  - Desktop object is **not created at container startup**
  - No constructor output for Desktop

Accessing a Lazy Bean Explicitly

- When the bean is explicitly requested:

```
Desktop desktop = (Desktop) context.getBean("desk");
```

- Behavior:
  - Desktop object is created at that moment
  - Output:

```
Desktop Object Created...
```

- Important:
  - Bean remains **singleton**
  - Created once and reused later

Lazy Singleton vs Prototype Scope

- **Prototype Scope**
  - Object created every time getBean() is called
- **Lazy Singleton**
  - Object created only once
  - Created **only when first requested**
- Lazy does **not change the scope**
  - It changes **when** the object is created

Lazy vs Eager Beans in the Application

- **Alien** → Eager (default)
- **Laptop** → Eager (default)
- **Desktop** → Lazy

Important Scenario: Dependency on Lazy Bean

- Scenario:
  - Alien is eager
  - Alien depends on Desktop
  - Desktop is marked lazy
- Configuration:

<property name="comp" ref="desk"/>

- Result:
  - Desktop object **is created**
- Reason:
  - A non-lazy bean depends on it
  - Spring must create it to satisfy dependency

Key Rule: Lazy Dependency Behavior

- If:
  - A **non-lazy (eager)** bean depends on a **lazy** bean
- Then:
  - The lazy bean **will still be created**
- Lazy initialization works **only if no eager bean depends on it**

Why Use Lazy Initialization

- Large applications may have:
  - Hundreds of beans
- Default behavior:
  - All singleton beans created at startup
- Drawbacks:
  - Increased startup time
  - Unnecessary memory usage
- Lazy initialization helps:
  - Improve performance
  - Load beans only when required
- Decision depends on:
  - Application requirements
  - Usage frequency of beans

Key Concepts Covered

- Singleton bean lifecycle
- Eager vs Lazy initialization
- lazy-init="true"
- Lazy singleton vs prototype
- Dependency impact on lazy beans

- Performance considerations

Summary

- Spring creates all singleton beans eagerly by default
- lazy-init="true" delays bean creation
- Lazy beans are created only when requested
- Lazy beans still behave as singletons
- Lazy initialization is overridden when an eager bean depends on it
- Useful for optimizing large Spring applications

Understanding Bean Type Specification Using getBean() in Spring

- When retrieving a bean from the Spring container using context.getBean("beanId"), the return type is **Object**.
- Because the return type is Object, **typecasting** is required to convert it into the expected class.

```
Alien alien1 = (Alien) context.getBean("alien1");
```

- This approach is not ideal because it introduces explicit casting and potential runtime errors.

Using getBean(String, Class<T>) to Avoid Typecasting

- Spring provides an overloaded version of getBean():

```
<T> T getBean(String name, Class<T> requiredType)
```

- This method allows specifying both:
  - The **bean ID**
  - The **expected class type**
- By doing this:
  - Spring returns an object of the specified type.
  - No explicit typecasting is required.

Example:

```
Alien alien = context.getBean("alien1", Alien.class);
```

- The returned object is already of type Alien.

Retrieving Beans by Type Instead of Name

- Spring also allows retrieving a bean **only by class type**, without specifying the bean ID.
- In this case, Spring searches for a bean **by type**.

Example:

```
Desktop desktop = context.getBean(Desktop.class);
```

- This works as long as:
  - Only **one bean of that type** exists in the container.

Search Mechanisms in Spring: By Name vs By Type

- Spring supports two primary lookup mechanisms:
  - **By Name**: context.getBean("beanId")
  - **By Type**: context.getBean(ClassName.class)
- When searching **by name**, there is no ambiguity.
- When searching **by type**, ambiguity arises if multiple beans share the same type or interface.

Using Interfaces with getBean()

- It is valid to retrieve a bean using an **interface type**.
- Interfaces also compile into .class files, so Spring can use them during lookup.

Example:

```
Computer computer = context.getBean(Computer.class);
```

- This works only if Spring can uniquely identify which implementation to return.

Problem: Multiple Beans Implementing the Same Interface

- If multiple beans implement the same interface:
  - Example: Laptop and Desktop both implement Computer
- Spring throws an error:
  - **Expected single matching bean but found two**
- This happens because Spring cannot decide which implementation to inject.

Resolving Bean Ambiguity Using primary="true"

- The ambiguity can be resolved by marking one bean as **primary**.
- The primary bean is preferred **only when Spring faces confusion** during type-based resolution.

XML Configuration Example:

```
<bean id="lap" class="org.springframeworkxmlbasedconfig.Laptop"
primary="true"/>
<bean id="desk" class="org.springframeworkxmlbasedconfig.Desktop"/>
```

- Now, when requesting:

```
Computer computer = context.getBean(Computer.class);
```

- Spring selects the Laptop bean because it is marked as **primary**.

Explicit Bean Reference Overrides primary

- If a bean is explicitly referenced by name:

```
Desktop desktop = context.getBean("desk", Desktop.class);
```

- Spring will return the specified bean **even if another bean is marked as primary**.
- The primary attribute is used **only when Spring must choose automatically**.

Best Practices for Bean Selection

- Use **bean names** when:
  - Multiple beans of the same type exist.
  - You want precise control and no ambiguity.
- Use **type-based lookup** when:
  - Only one bean of that type exists.
  - Or one bean is clearly marked as primary.

Key Takeaways

- getBean(String) returns an Object, requiring typecasting.
- getBean(String, Class<T>) returns a strongly typed object.
- getBean(Class<T>) performs a **by-type search**.
- Multiple beans of the same type cause ambiguity.
- primary="true" resolves ambiguity during type-based injection.
- Explicit bean names always override primary.

Relevant Code Summary

```
// By name and type
Desktop desktop = context.getBean("desk", Desktop.class);

// By type only
Desktop desktopByType = context.getBean(Desktop.class);

// Interface-based retrieval
```
Computer computer = context.getBean(Computer.class); // Works due to primary bean

Understanding Inner Beans in Spring XML Configuration

- This section explains the concept of **inner beans** in the **Spring Framework** using XML-based configuration.

- Before introducing inner beans, unused bean definitions (such as Desktop) are commented out to simplify the setup and focus only on the required dependency.

Dependency Scenario: Alien and Computer

- The Alien class depends on a Computer.
- Laptop is the concrete implementation of the Computer interface.
- Initially:
  - Laptop is defined as a standalone bean.
  - Alien references the Laptop bean using ref.

Implication:

- A standalone Laptop bean becomes part of the **entire Spring container**.
- Any other bean (e.g., Human, Car) can reuse the same Laptop instance by referencing it.

Problem: Limiting Bean Scope to a Single Bean

- In some cases, the Laptop bean:
  - Is created **only to satisfy the dependency of Alien**
  - Is **not intended** to be reused by other beans
- Making it a global bean exposes it unnecessarily across the application.

Solution: Using Inner Beans

- An **inner bean** is a bean defined **inside another bean's property**.
- It is:
  - Scoped only to the enclosing (outer) bean
  - Not accessible or reusable by other beans in the container

---

How to Define an Inner Bean

- Remove the ref attribute from the property.
- Change the property to use opening and closing tags.
- Define the dependent bean **inside the <property> tag**.

Example Configuration:

```
<bean id="alien1"
```

```
class="org.springframeworkxmlbasedconfig.Alien"
autowire="byType">
    <property name="age" value="28"/>
    <property name="comp">
        <bean id="lap"
class="org.springframeworkxmlbasedconfig.Laptop"
primary="true"/>
    </property>
</bean>
```

Outer Bean vs Inner Bean

- **Outer Bean**
  - alien1
  - Managed by the Spring container and accessible throughout the application
- **Inner Bean**
  - Laptop defined inside the comp property
  - Can only be used by alien1
  - Cannot be referenced by other beans

Behavior and Validation

- The application runs successfully with the inner bean configuration.
- Dependency injection still works as expected.
- The Laptop instance is created and injected only for Alien.

Key Characteristics of Inner Beans

- Defined inside a <property> or <constructor-arg>
- No need to be referenced using ref
- Not available for lookup via getBean()
- Lifecycle is tied to the outer bean

Design Consideration

- Choosing between:
  - **Standalone beans** (reusable across the application)
  - **Inner beans** (restricted to a specific dependency)

- This is a design decision based on:
  - Reusability needs
  - Encapsulation requirements
  - Application architecture

Summary

- Inner beans are used to restrict a dependency to a single bean.
- They help avoid unnecessary exposure of beans across the application.
- Defined directly inside the dependent bean’s property.
- Ideal when a bean exists only to serve one specific dependency.