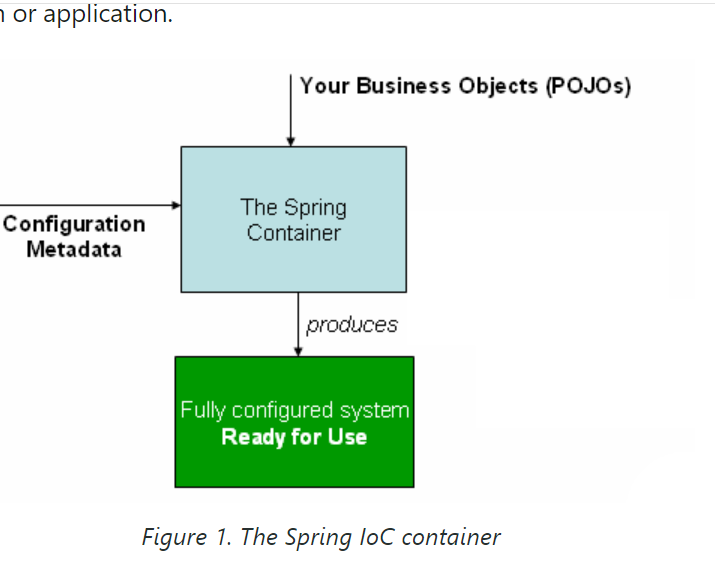
**7 Exploring Spring Framework**

**The IoC Container**

**Introduction to the Spring IoC Container and Beans**

* **Inversion of Control (IoC) Container**:
  + This chapter covers Spring's implementation of the Inversion of Control (IoC) principle, also known as Dependency Injection (DI).
  + IoC is a process where objects define their dependencies (the other objects they work with) only through constructor arguments, factory method arguments, or properties set on the object instance after it is constructed or returned from a factory method.
  + The container then injects these dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern.
* **org.springframework.beans and org.springframework.context Packages**:
  + These packages are the basis for Spring Framework's IoC container.
  + The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object.
  + ApplicationContext is a sub-interface of BeanFactory and adds more enterprise-specific functionality, such as:
    - Easier integration with Spring's AOP features
    - Message resource handling for use in internationalization
    - Event publication
    - Application-layer specific contexts such as the WebApplicationContext for use in web applications.
  + In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory and is used exclusively in this chapter in descriptions of Spring's IoC container.
* **Beans**:
  + In Spring, the objects that form the backbone of your application and are managed by the Spring IoC container are called beans.
  + A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.
  + Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

**Container Overview**

* **ApplicationContext Interface**:
  + The org.springframework.context.ApplicationContext interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans.
  + The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. This configuration metadata is represented in XML, Java annotations, or Java code.
  + It lets you express the objects that compose your application and the rich interdependencies between those objects.
* **Common Implementations**:
  + Several implementations of the ApplicationContext interface are supplied with Spring. In standalone applications, it is common to create an instance of ClassPathXmlApplicationContext or FileSystemXmlApplicationContext.
  + While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.
* **Application Scenarios**:
  + In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the web.xml file of the application typically suffices.
  + If you use the Spring Tools for Eclipse (an Eclipse-powered development environment), you can easily create this boilerplate configuration with a few mouse clicks or keystrokes.
* **Diagram Overview**:
  + The following diagram shows a high-level view of how Spring works. Your application classes are combined with configuration metadata so that, after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.
  + **Figure 1. The Spring IoC container**

**Configuration Metadata**

**XML-based Configuration**

* **Purpose**:
  + As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.
* **Formats**:
  + XML-based metadata is traditionally supplied in a simple and intuitive format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.
  + XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose Java-based configuration for their Spring applications.

**Java-based Configuration**

* **Annotations**:
  + For information about using other forms of metadata with the Spring container, see:
    - Annotation-based configuration: Define beans using annotation-based configuration metadata.
    - Java-based configuration: Define beans external to your application classes by using Java rather than XML files. To use these features, see the @Configuration, @Bean, @Import, and @DependsOn annotations.

**Bean Definitions**

* **Typical Configuration**:
  + Spring configuration consists of at least one and typically more than one bean definition that the container must manage.
  + XML-based configuration metadata configures these beans as <bean/> elements inside a top-level <beans/> element.
  + Java configuration typically uses @Bean-annotated methods within a @Configuration class.
* **Actual Objects**:
  + These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, persistence layer objects such as repositories or data access objects (DAOs), presentation objects such as web controllers, infrastructure objects such as a JPA EntityManagerFactory, JMS queues, and so forth.
  + Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of repositories and business logic to create and load domain objects.

**Configuration Metadata**

**XML-Based Configuration Metadata**

* **Decoupling from Configuration Format**:
  + The Spring IoC container is totally decoupled from the format in which the configuration metadata is written.
  + XML-based metadata is not the only allowed form of configuration metadata.
* **Modern Preferences**:
  + Many developers now prefer Java-based configuration for their Spring applications.

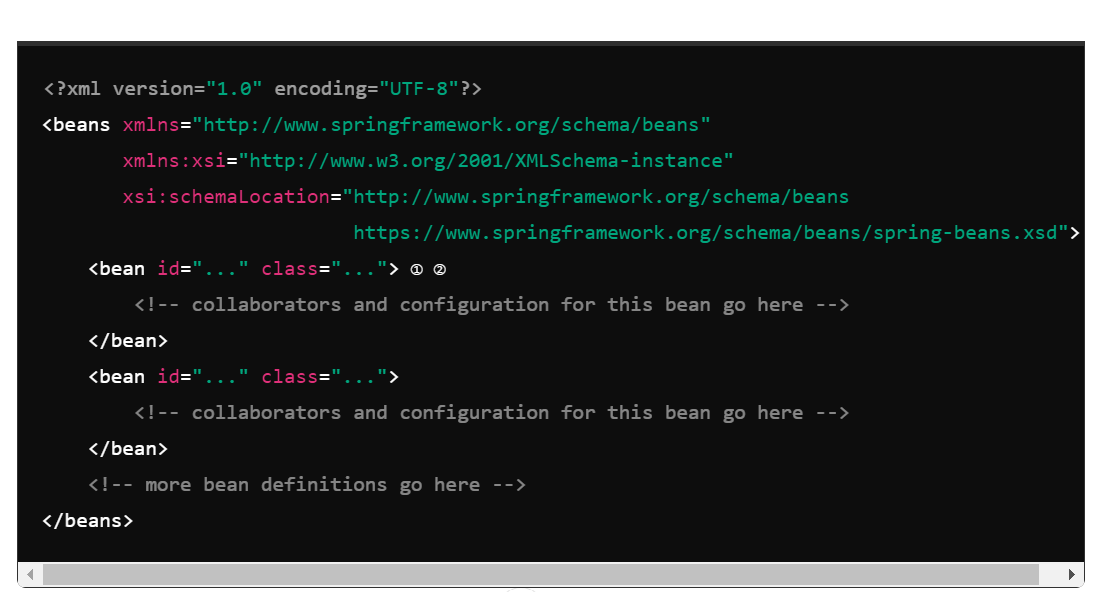
**Other Forms of Metadata**

* **Annotation-Based Configuration**:
  + Define beans using annotation-based configuration metadata.
* **Java-Based Configuration**:
  + Define beans external to your application classes using Java rather than XML files.
  + Key annotations include:
    - @Configuration
    - @Bean
    - @Import
    - @DependsOn

**Bean Definitions**

* **Spring Configuration**:
  + Consists of at least one and typically more than one bean definition that the container must manage.
  + XML-based configuration metadata configures these beans as <bean/> elements inside a top-level <beans/> element.
  + Java configuration typically uses @Bean-annotated methods within a @Configuration class.
* **Types of Beans**:
  + Bean definitions correspond to the actual objects that make up your application.
  + Typically include service layer objects, persistence layer objects (such as repositories or data access objects (DAOs)), presentation objects (such as Web controllers), and infrastructure objects (such as a JPA EntityManagerFactory, JMS queues, etc.).
  + Fine-grained domain objects are usually not configured in the container as it is usually the responsibility of repositories and business logic to create and load domain objects.

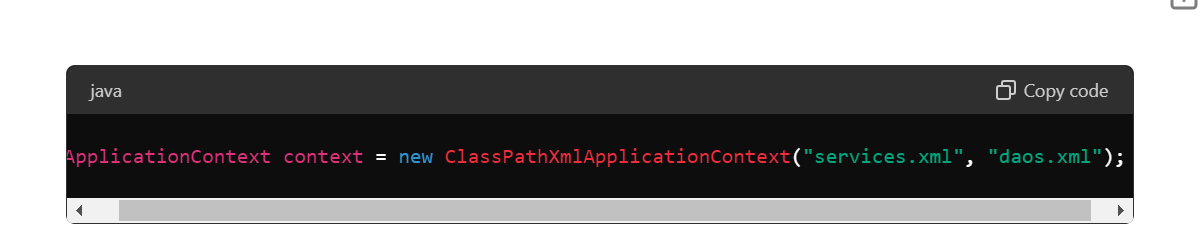
**Basic Structure of XML-Based Configuration Metadata**

****

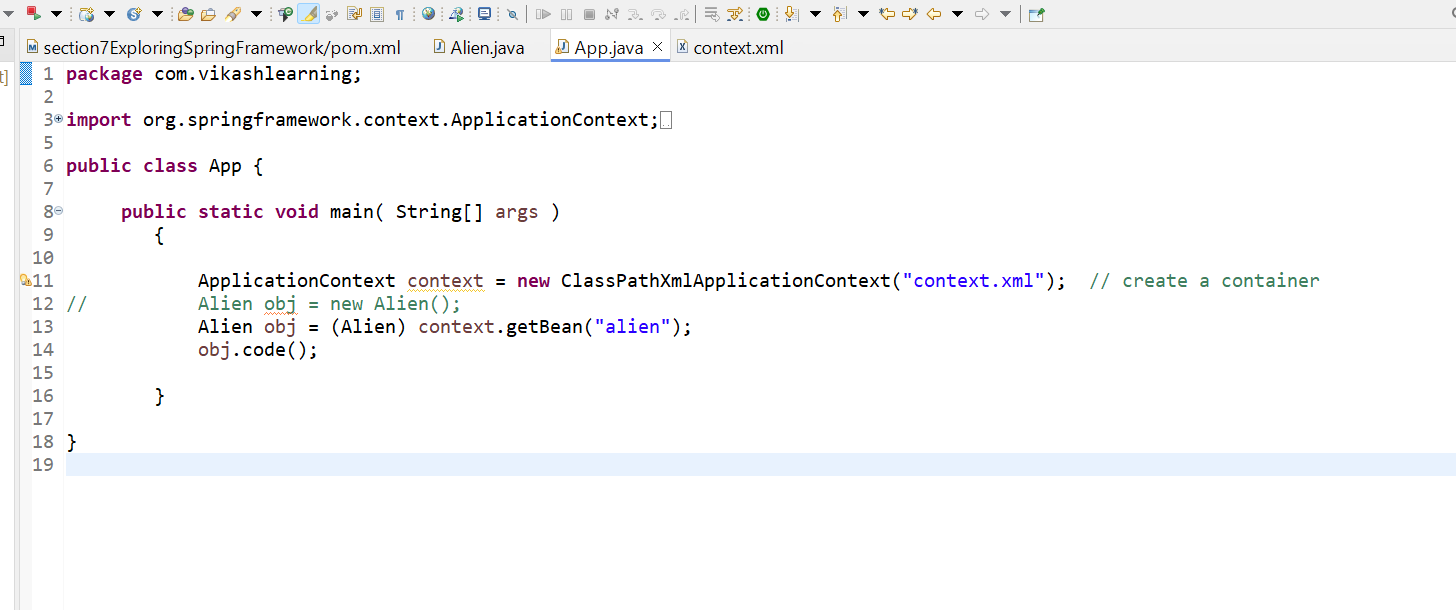
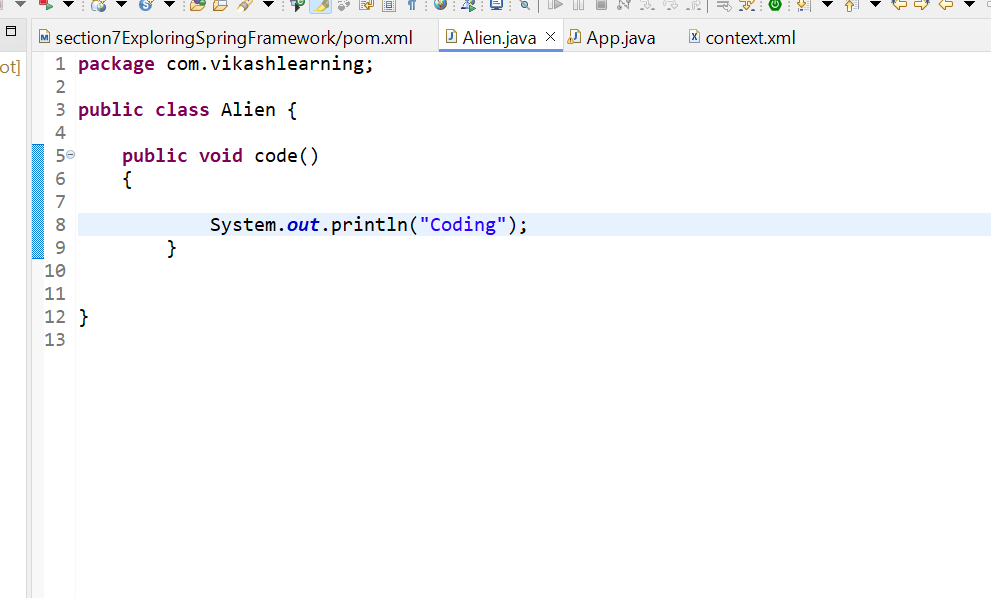
* **Attributes**:
  + ① id: A string that identifies the individual bean definition.
  + ② class: Defines the type of the bean and uses the fully qualified class name.
* **Referring to Collaborating Objects**:
  + The value of the id attribute can be used to refer to collaborating objects.
  + XML for referring to collaborating objects is not shown in this example. See Dependencies for more information.

**Instantiating a Container**

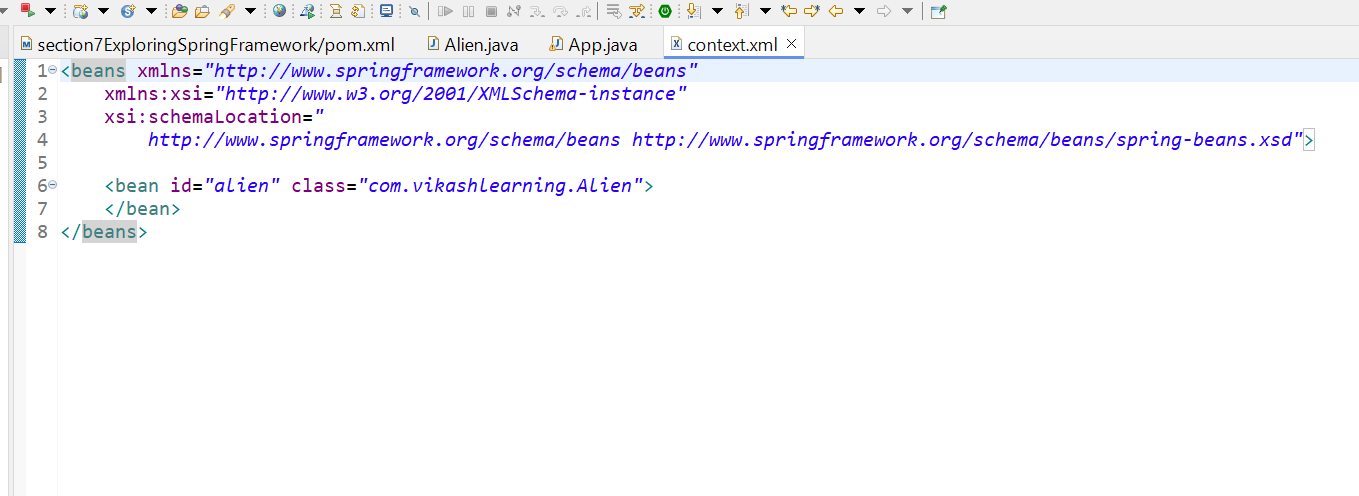
* **Resource Strings**:
  + The location path or paths supplied to an ApplicationContext constructor are resource strings that let the container load configuration metadata from various external resources, such as the local file system, the Java CLASSPATH, and so on.
* **Example in Java**:

****

**Explanation**:

* The ClassPathXmlApplicationContext constructor looks for files in the classpath as specified in the parameters.
* This line of code indicates which classes need to have objects created by the IoC container.
* As this ClassPathXmlApplicationContext(); it will look file from the class path like we have to put our file in the parameter.
* 
* 

Context.xml



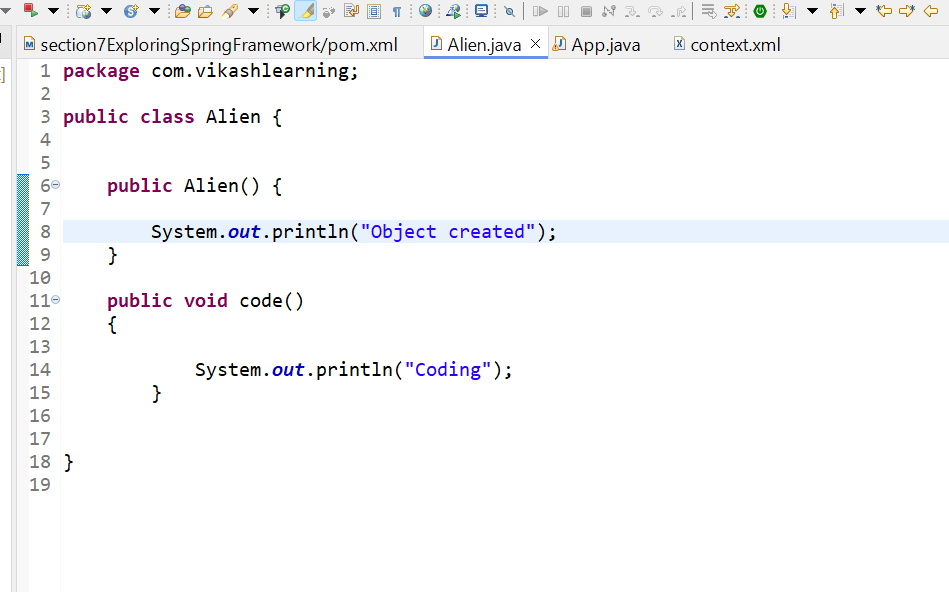
**Objects**

Now the Question is

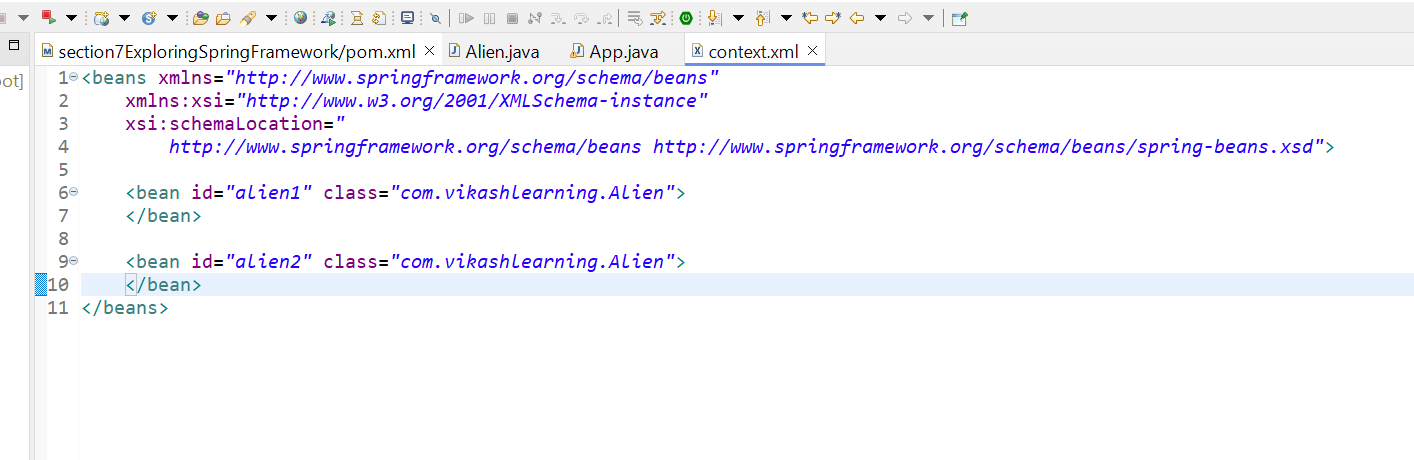
1Question: - At which line object get created.

Answer:

So if we create one constructor of Alien class and comment.


And the application it will print “Object created” means   
  
Object gets created in  
 ApplicationContext context = **new** ClassPathXmlApplicationContext("context.xml"); // create a container  
  
and container as well and decide which all property have bean tag create object of all.

Question 2 What if we will add two beans of same class with diff id and run the App class.  


Answer : it will print “Object created two time   
  
Object created

Object created

### Spring Bean Scopes

### 

#### Overview

* **Spring Framework** supports several **bean scopes** to define how Spring manages bean instances.
* Scopes determine the **lifecycle** and **availability** of bean instances within the **Spring IoC container**.

#### Supported Scopes

1. **Singleton**
   * **Description:** Default scope where a single instance of the bean is created per Spring IoC container.
   * **Characteristics:**
     + One shared instance per container.
     + Cached instance used for subsequent requests.
     + Default scope if not specified explicitly.
2. **Prototype**
   * **Description:** A new bean instance created each time it's requested.
   * **Characteristics:**
     + Any number of instances.
     + Not cached; new instance per request.
     + Useful for stateful beans or where frequent re-instantiation is required.

#### Web-aware Scopes

* These scopes are available in web-aware applications (WebApplicationContext):

1. **Request**
   * **Description:** Scoped to the lifecycle of an HTTP request.
   * **Characteristics:**
     + New instance for each HTTP request.
     + Valid within a web context (WebApplicationContext).
2. **Session**
   * **Description:** Scoped to the lifecycle of an HTTP session.
   * **Characteristics:**
     + One instance per HTTP session.
     + Valid within a web context (WebApplicationContext).
3. **Application**
   * **Description:** Scoped to the lifecycle of a ServletContext.
   * **Characteristics:**
     + One instance per ServletContext.
     + Valid within a web context (WebApplicationContext).
4. **Websocket**
   * **Description:** Scoped to the lifecycle of a WebSocket.
   * **Characteristics:**
     + One instance per WebSocket connection.
     + Valid within a web context (WebApplicationContext).

#### Custom Scopes

* Developers can define **custom scopes** beyond the built-in ones to meet specific application requirements.

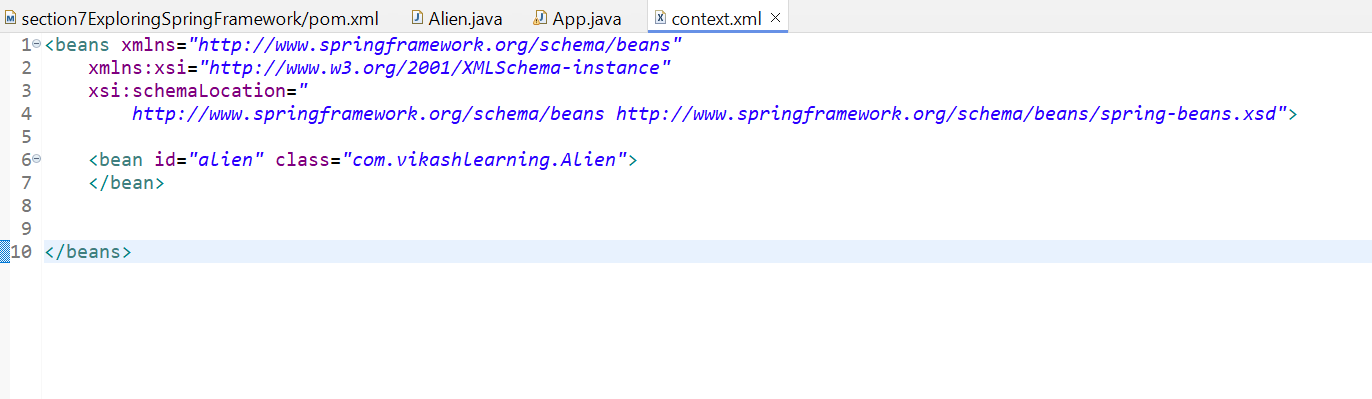
#### Singleton vs Prototype

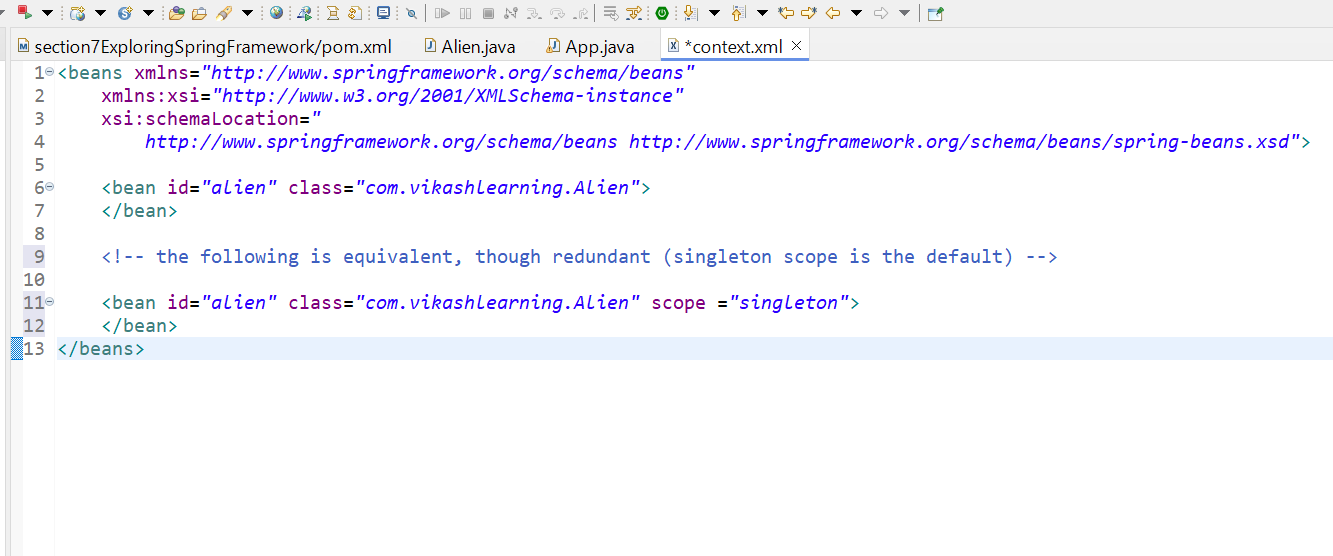
* **Singleton:** One shared instance per container, cached for efficiency.
* **Prototype:** New instance created each time it's requested, suitable for non-thread-safe or mutable beans.

#### Differences from GoF Singleton Pattern

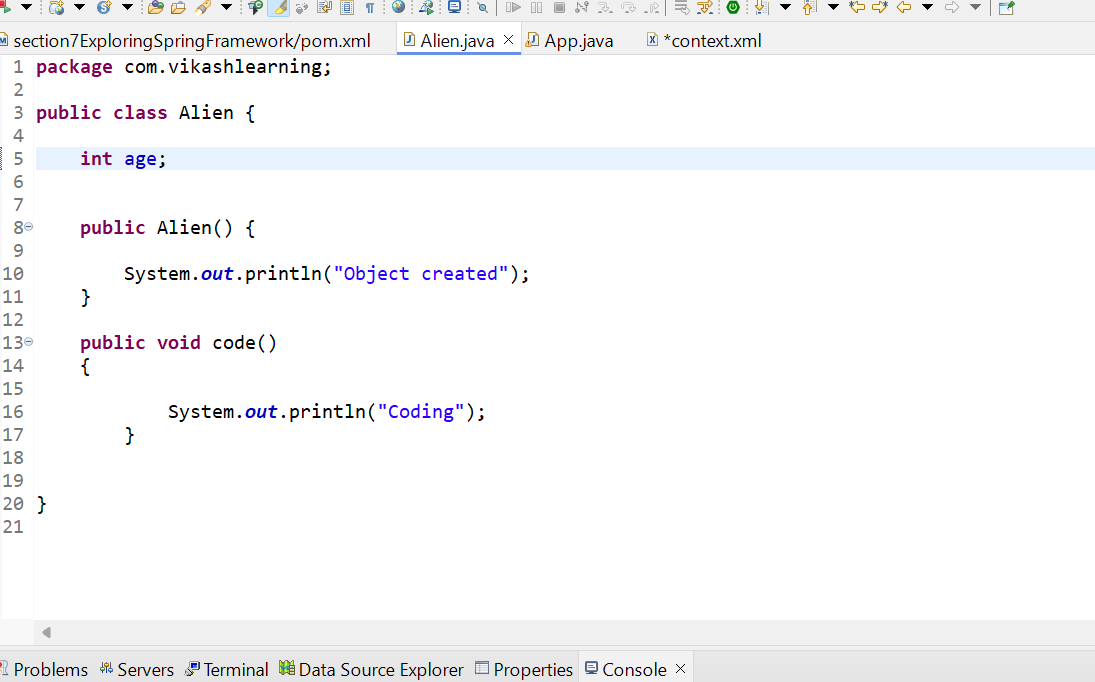
* Spring singleton scope is **per-container** and **per-bean**, different from the GoF singleton pattern which is **per-ClassLoader**.

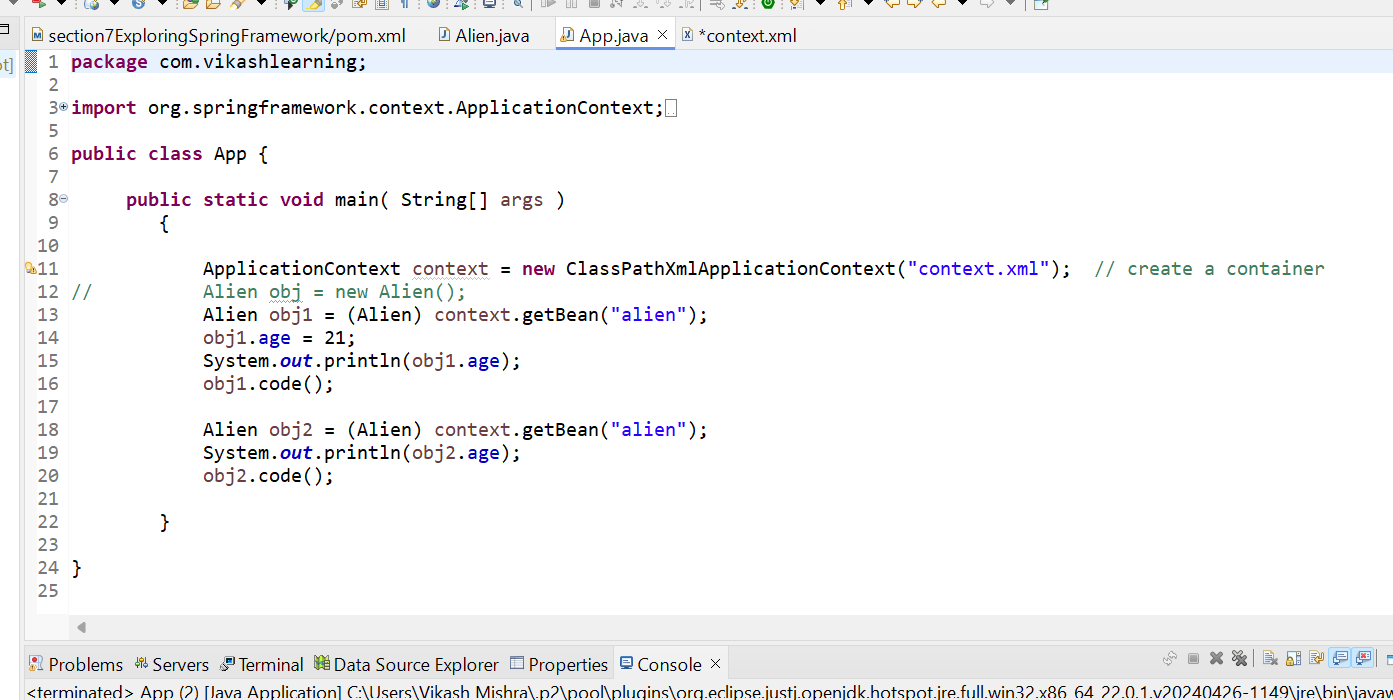
These notes provide a comprehensive overview of Spring bean scopes, detailing their definitions, characteristics, and usage contexts, including differences from traditional design patterns.

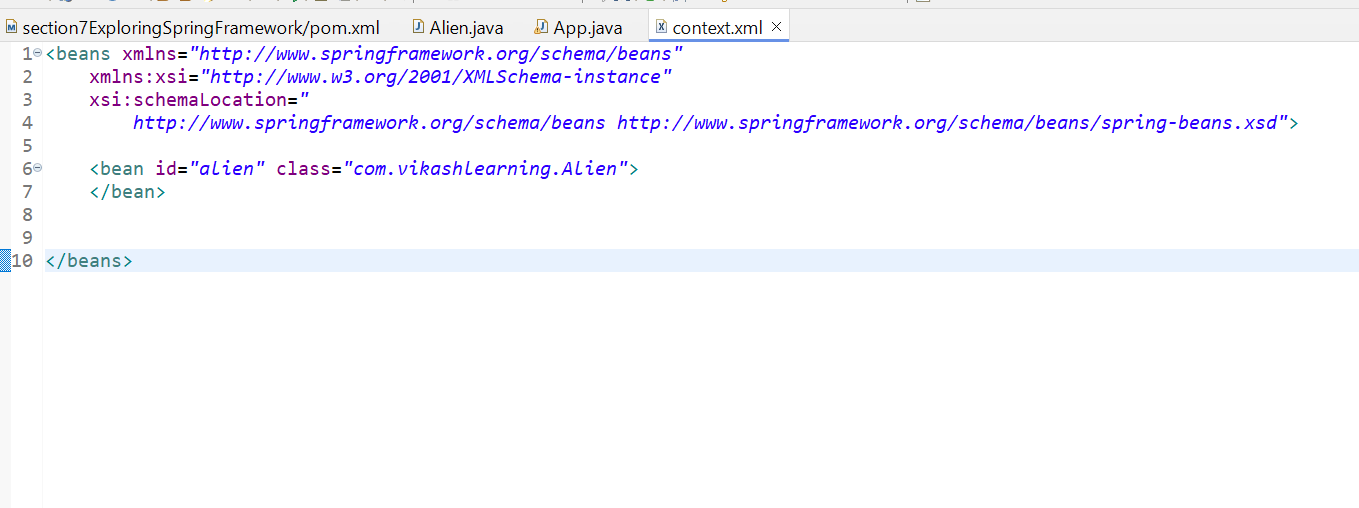




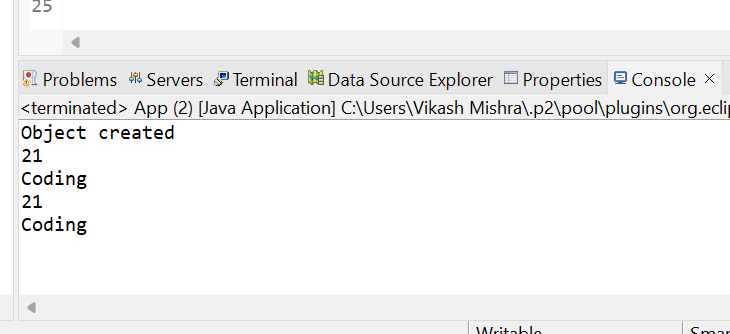
Example;







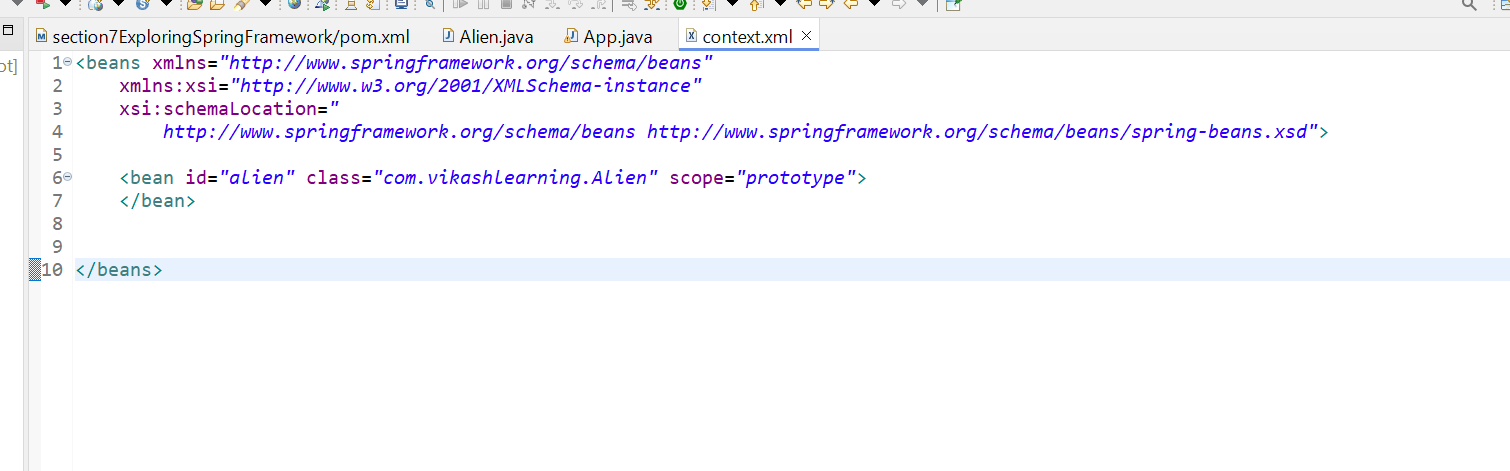
O/P



Means how many times we create objects it will create only one time because the default scope is singleton.

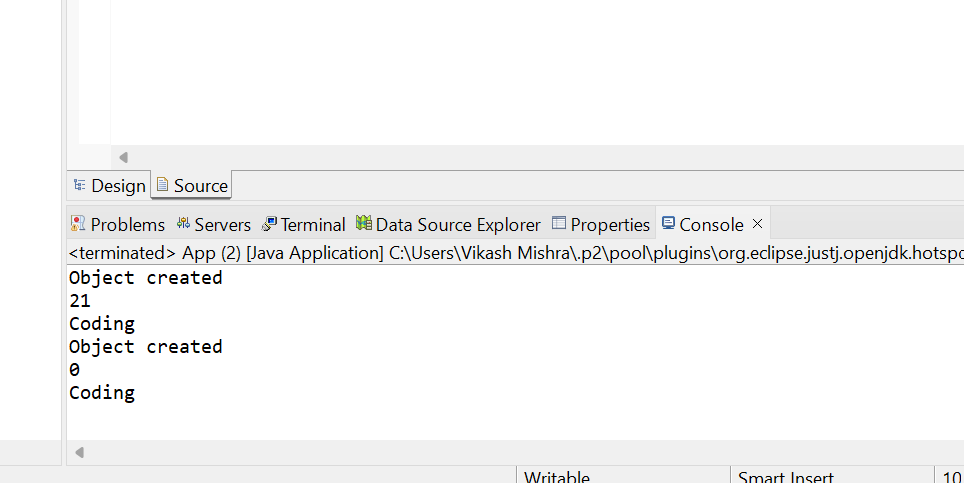
Question What if we will run same code keeping scope prototype.

Answer: After making scope prototype and run the App.java



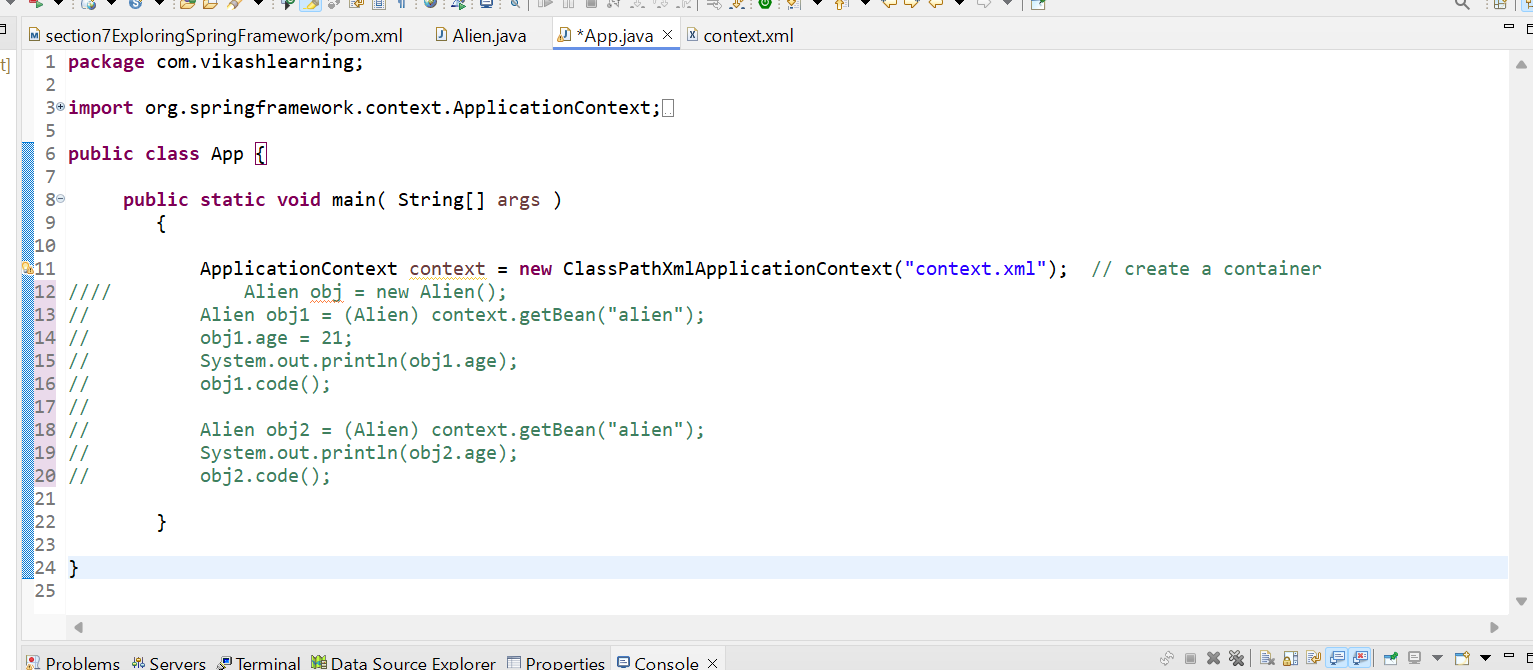
O/P

you can see the output. The Object is created two times.



Question: What if we will comment all the line and run

Below code



Answer;

You can see there is no output. Because Object get created on we call. getbean() method earlier why it was happening because the default scope is Singlton and it create object once container is load.



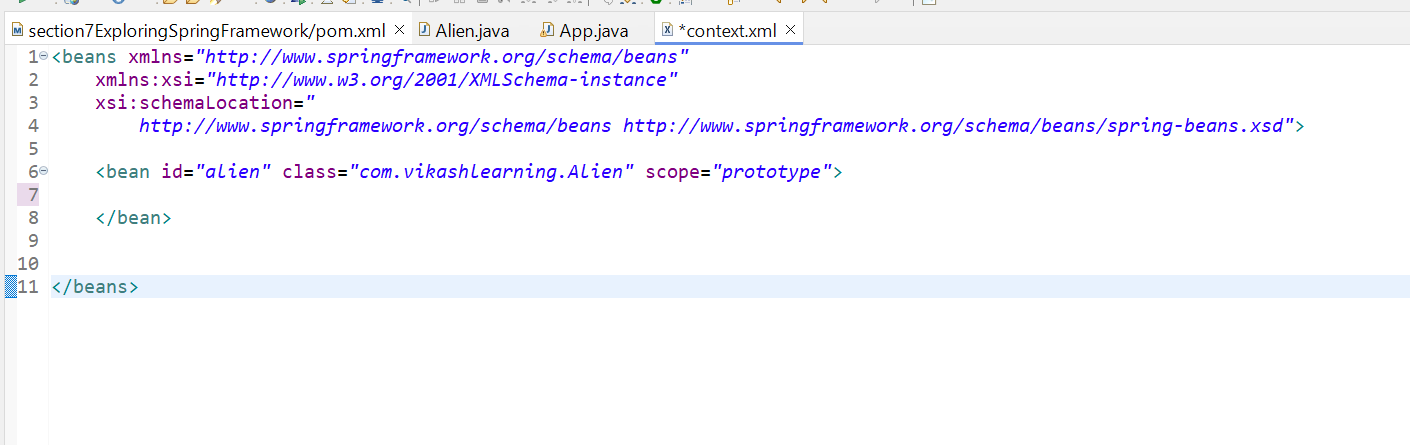
**Setter Injection**

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument static factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected by using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

Normal Approach





By Setter Injection

Alien.Java

**package** com.vikashlearning;

**public** **class** Alien {

**private** **int** age;

**public** Alien() {

System.***out***.println("Object created");

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

System.***out***.println("Setter is called ");

**this**.age = age;

}

**public** **void** code()

{

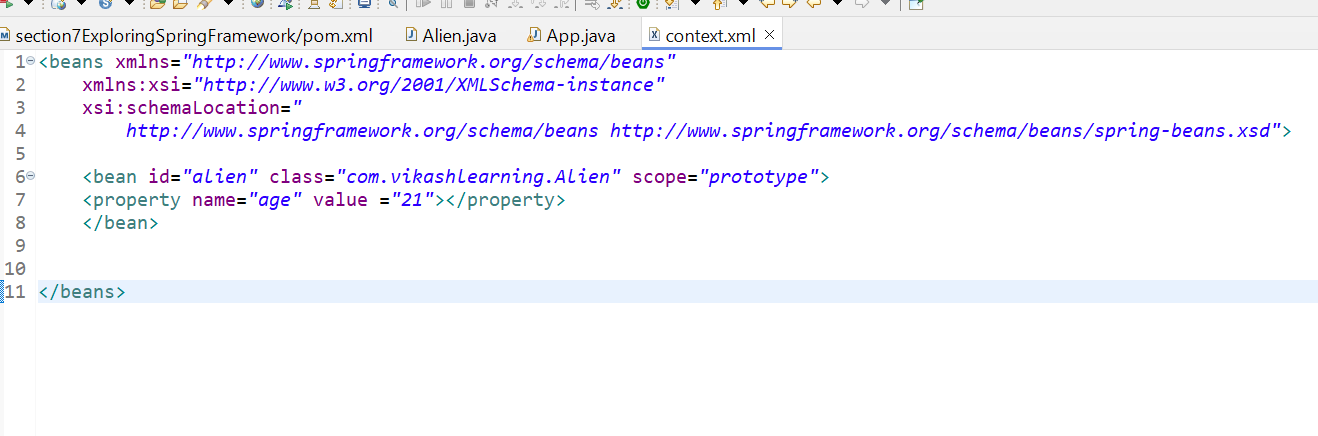
System.***out***.println("Coding");

}

}

By Dependency Injection.

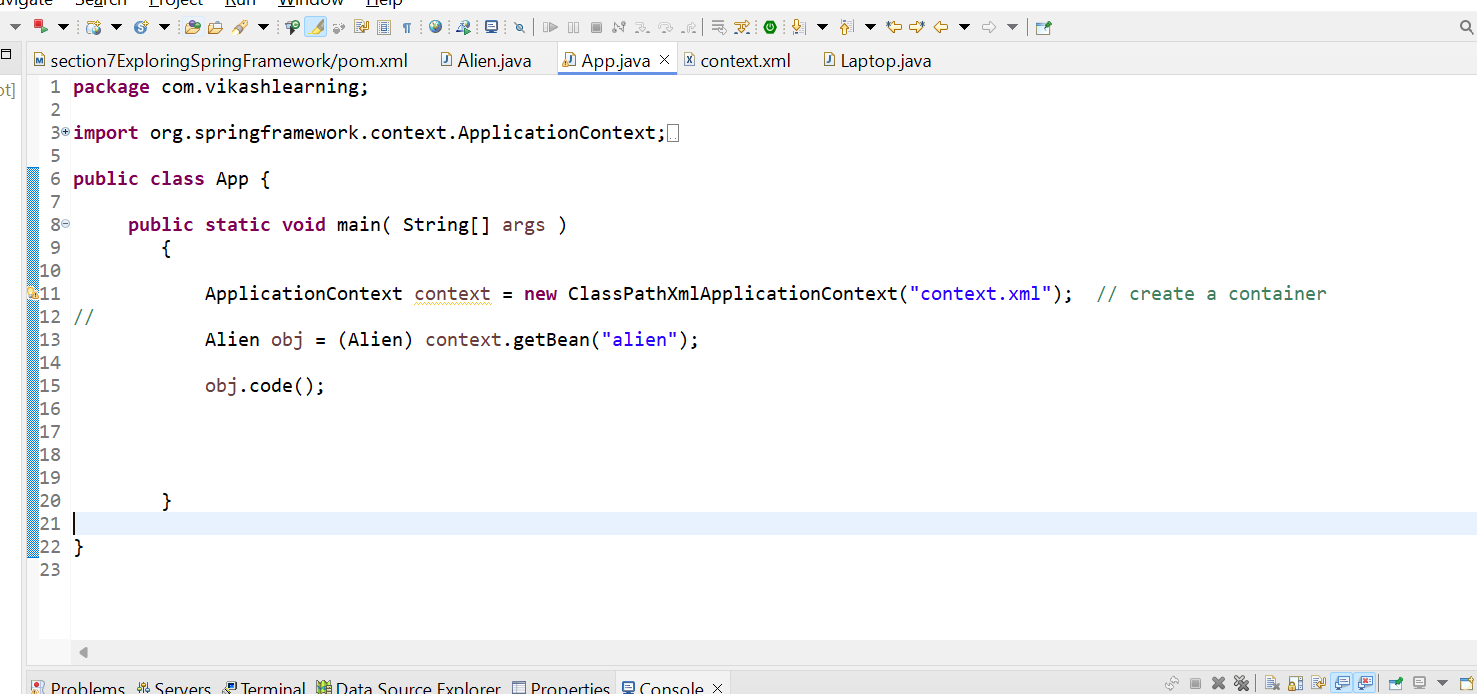
When value is primitive

****

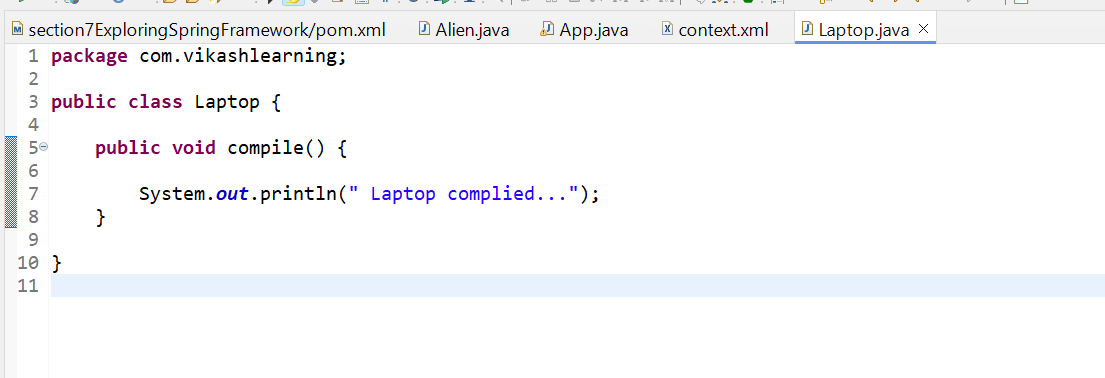
****

When Value is non primitive

App.jAVA



Laptop.java



Alien.java

**package** com.vikashlearning;

**public** **class** Alien {

**private** **int** age;

**private** Laptop laptop;

**public** Laptop getLaptop() {

**return** laptop;

}

**public** **void** setLaptop(Laptop laptop) {

**this**.laptop = laptop;

}

**public** Alien() {

System.***out***.println("Object created");

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

System.***out***.println("Setter is called ");

**this**.age = age;

}

**public** **void** code()

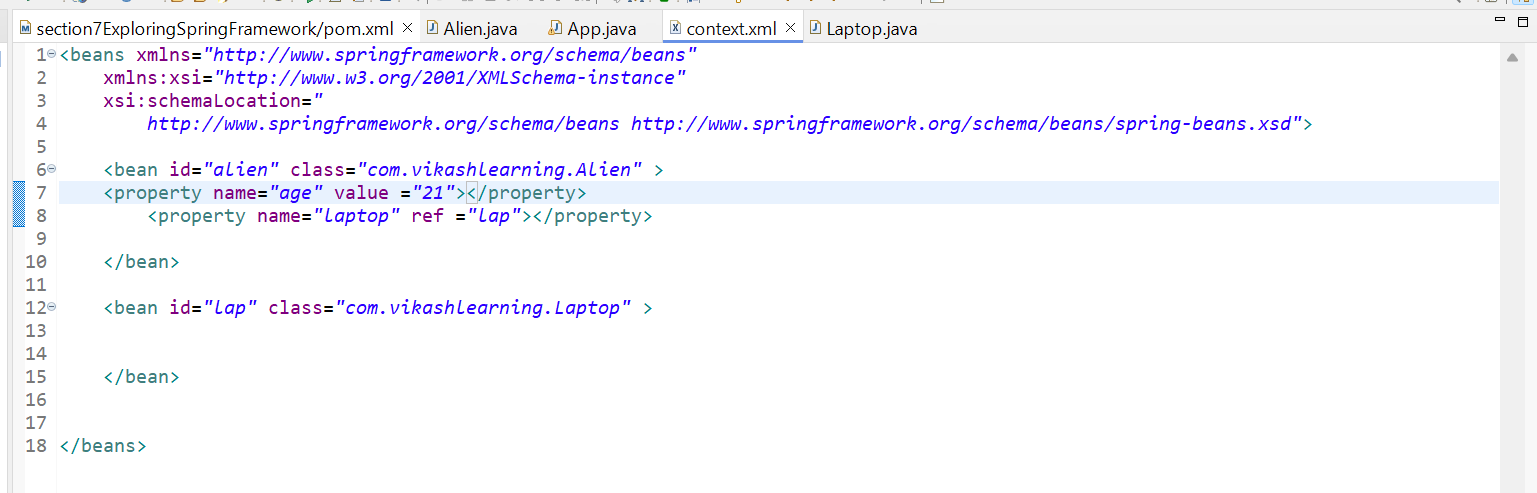
{

System.***out***.println("Coding");

laptop.compile();

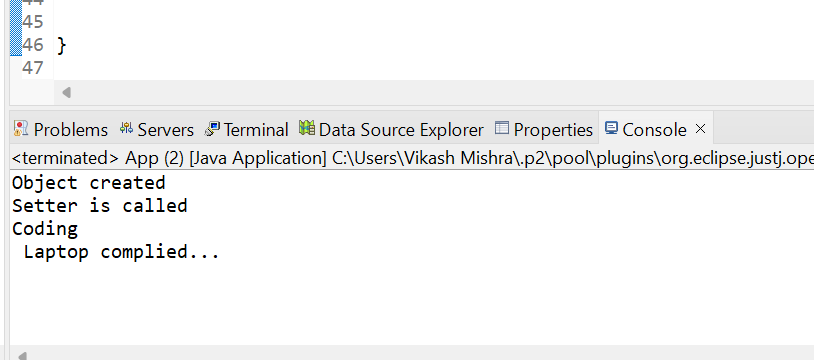
}

}



After running the App.java app

O/P



## Constructor-based Dependency Injection

**Constructor-Based DI** involves providing dependencies to a bean through its constructor. Spring uses the constructor to create the bean and inject all required dependencies at once.

Example   
  
**package** com.vikashlearningConstructerInjection;

**public** **class** AlienConstructorInj {

**private** **int** age;

**private** LaptopConstructorInj laptop;

**public** LaptopConstructorInj getLaptop() {

**return** laptop;

}

**public** **void** setLaptop(LaptopConstructorInj laptop) {

**this**.laptop = laptop;

}

**public** AlienConstructorInj() {

System.***out***.println("Object created");

}

**public** AlienConstructorInj(**int** age ) {

System.***out***.println("Parameterzied constructor is getting called");

**this**.age=age;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

System.***out***.println("Setter is called ");

**this**.age = age;

}

**public** **void** code()

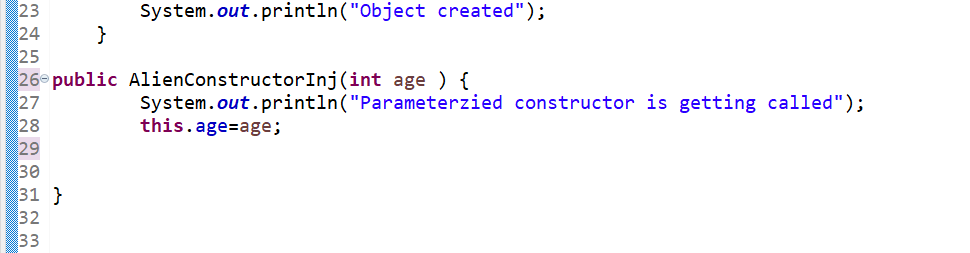
{

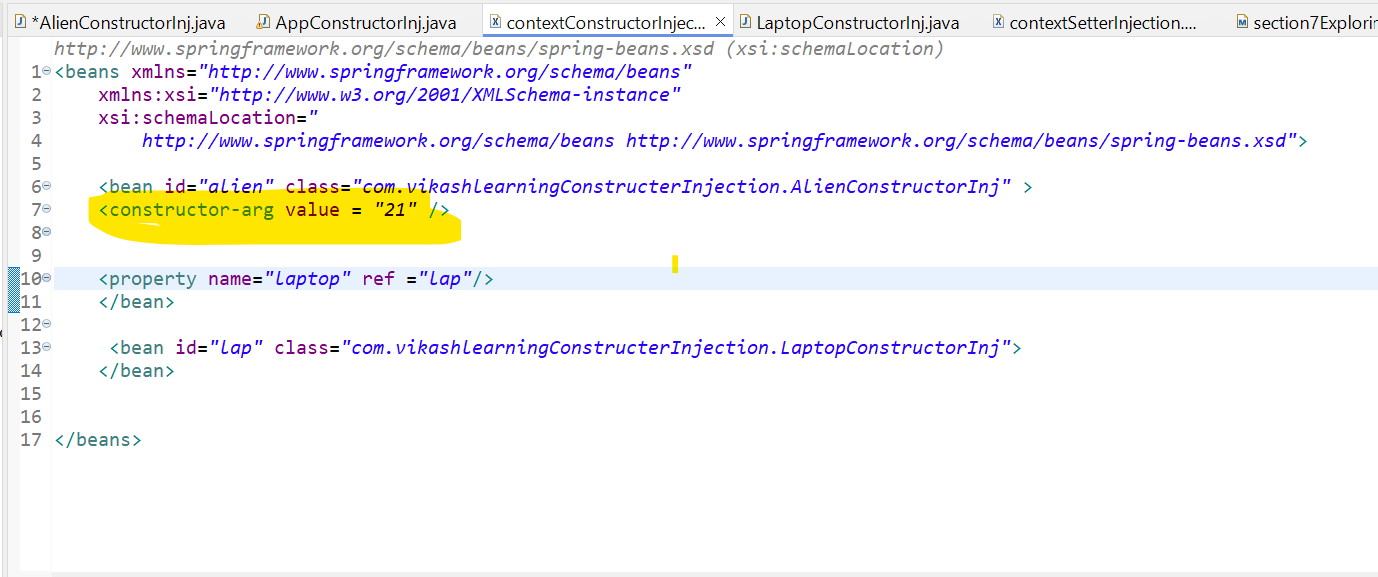
System.***out***.println("Coding");

laptop.compile();

}

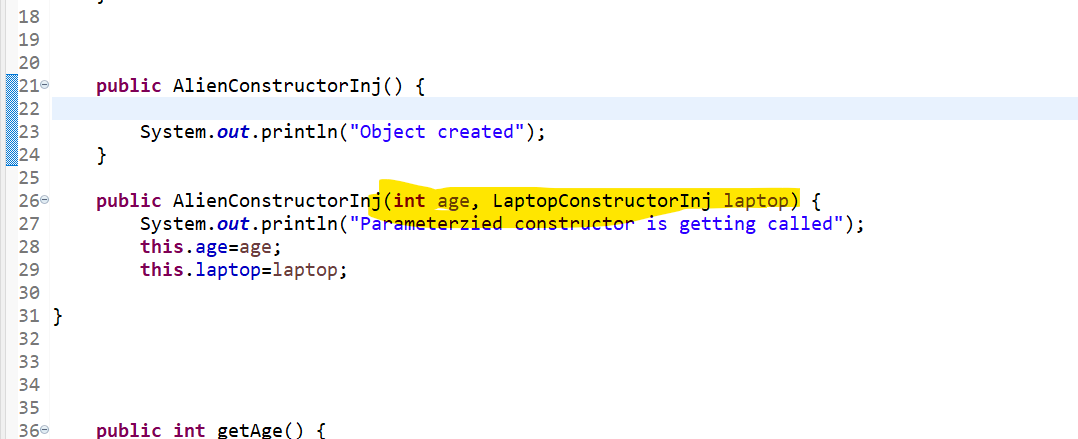
}

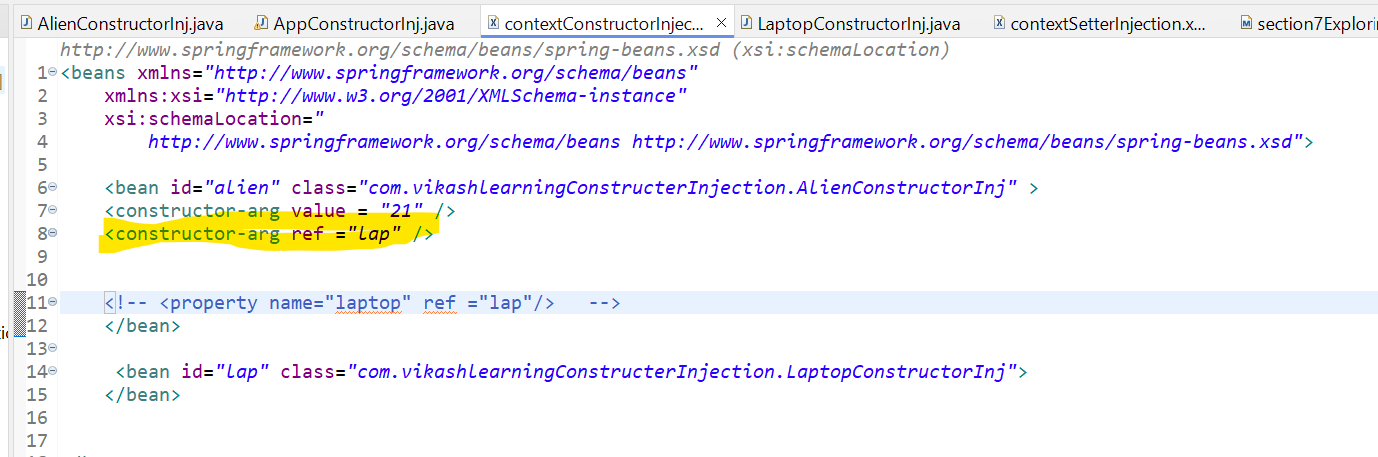




**. Constructor Argument Resolution:**

* **Single Dependency:**
  + Scenario: When there is only one constructor argument, Spring easily resolves it by passing the specified bean.  
      
    Refer the previous example
* **Multiple Dependencies:**
  + Scenario: If a bean has multiple constructor arguments, Spring matches the arguments by type. If multiple constructors are available, Spring uses the one that matches the provided arguments.

Example:  
  




**Note:**

**Parameter Sequence Matching:**

* **Default Behavior:** By default, Spring resolves constructor arguments based on the order of the <constructor-arg> elements in the XML configuration.
* **Error Scenario:** If the sequence of constructor arguments in the bean definition does not match the sequence of constructor parameters in the class, it can lead to errors. For instance, if a constructor expects parameters in a different order than provided in the XML, Spring will not be able to correctly match them.

**Attribute to Resolve Sequence Issues:**

* **<constructor-arg> Attributes:**
  + **type Attribute:** You can explicitly specify the type of the constructor argument using the type attribute if there are multiple constructors or if the parameter order is ambiguous. This helps Spring identify which constructor to use.

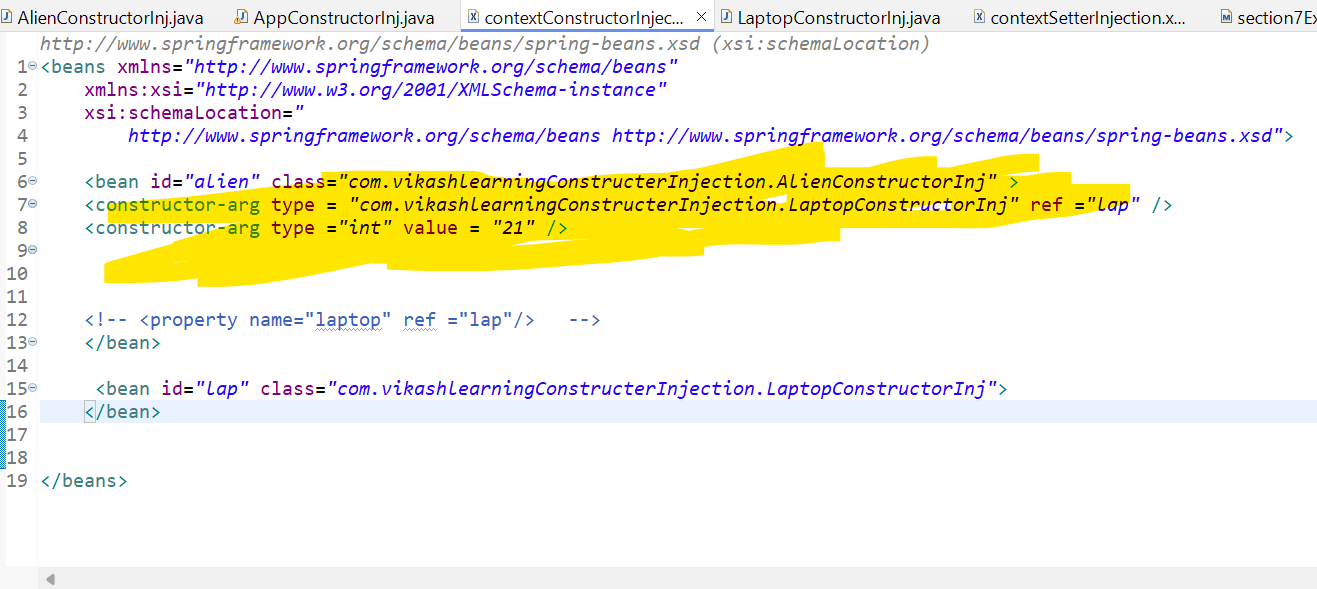
**You can see there is diff between sequence of constructor parameter and injection in xml file so we have mentioned type attribute**

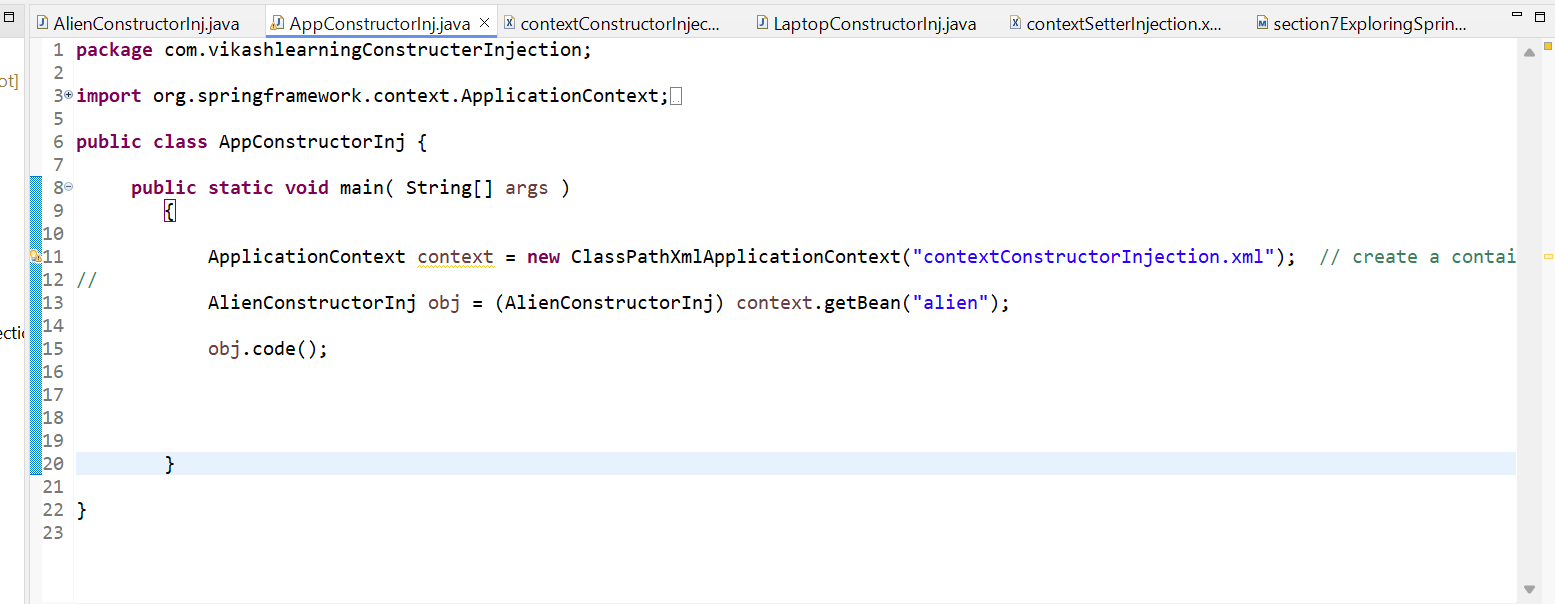
* **package** com.vikashlearningConstructerInjection;
* **public** **class** AlienConstructorInj {
* **private** **int** age;
* **private** LaptopConstructorInj laptop;

* **public** LaptopConstructorInj getLaptop() {
* **return** laptop;
* }
* **public** **void** setLaptop(LaptopConstructorInj laptop) {
* **this**.laptop = laptop;
* }
* **public** AlienConstructorInj() {
* System.***out***.println("Object created");
* }
* **public** AlienConstructorInj(**int** age, LaptopConstructorInj laptop) {
* System.***out***.println("Parameterzied constructor is getting called");
* **this**.age=age;
* **this**.laptop=laptop;
* }

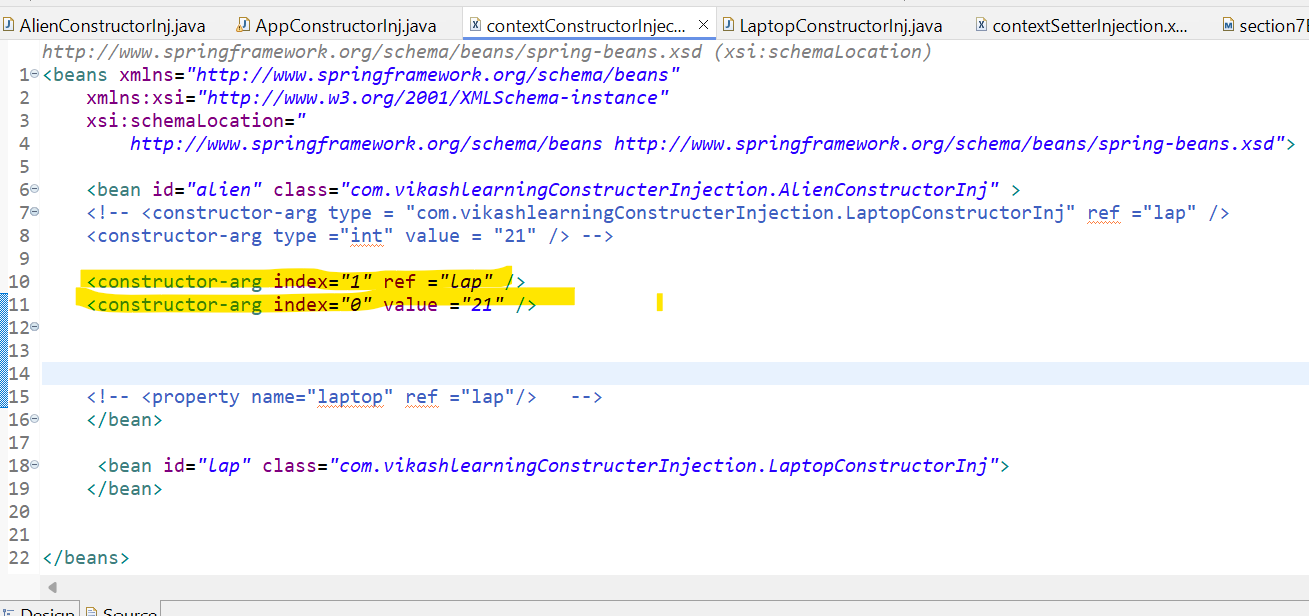


* **public** **int** getAge() {
* **return** age;
* }
* **public** **void** setAge(**int** age) {
* System.***out***.println("Setter is called ");
* **this**.age = age;
* }
* **public** **void** code()
* {
* System.***out***.println("Coding");
* laptop.compile();
* }
* }





* + **index Attribute:** In cases where multiple constructors are available, you can use the index attribute to specify the exact position of the argument in the constructor.



**Creating Interface**

Suppose you have an application that requires sending notifications to users through various channels like email, SMS, and push notifications. You want your application to be flexible, allowing you to easily switch between different notification implementations or add new ones in the future.

To achieve this, you can define an interface called *NotificationService* that provides a contract for sending notifications:

public interface NotificationService {

void sendNotification(String message, String recipient);

}

Next, you can create different implementations of the *NotificationService* interface for each channel. For example, let's create an *EmailNotificationService* and an *SMSNotificationService*:

public class EmailNotificationService implements NotificationService {

public void sendNotification(String message, String recipient) {

// Logic to send an email notification

}

}

public class SMSNotificationService implements NotificationService {

public void sendNotification(String message, String recipient) {

// Logic to send an SMS notification

}

}

Now, let's say you have a class called *NotificationSender* that needs to send notifications. Instead of directly depending on concrete implementations like *EmailNotificationService* or *SMSNotificationService*, you can depend on the *NotificationService* interface:

public class NotificationSender {

private final NotificationService notificationService;

public NotificationSender(NotificationService notificationService) {

this.notificationService = notificationService;

}

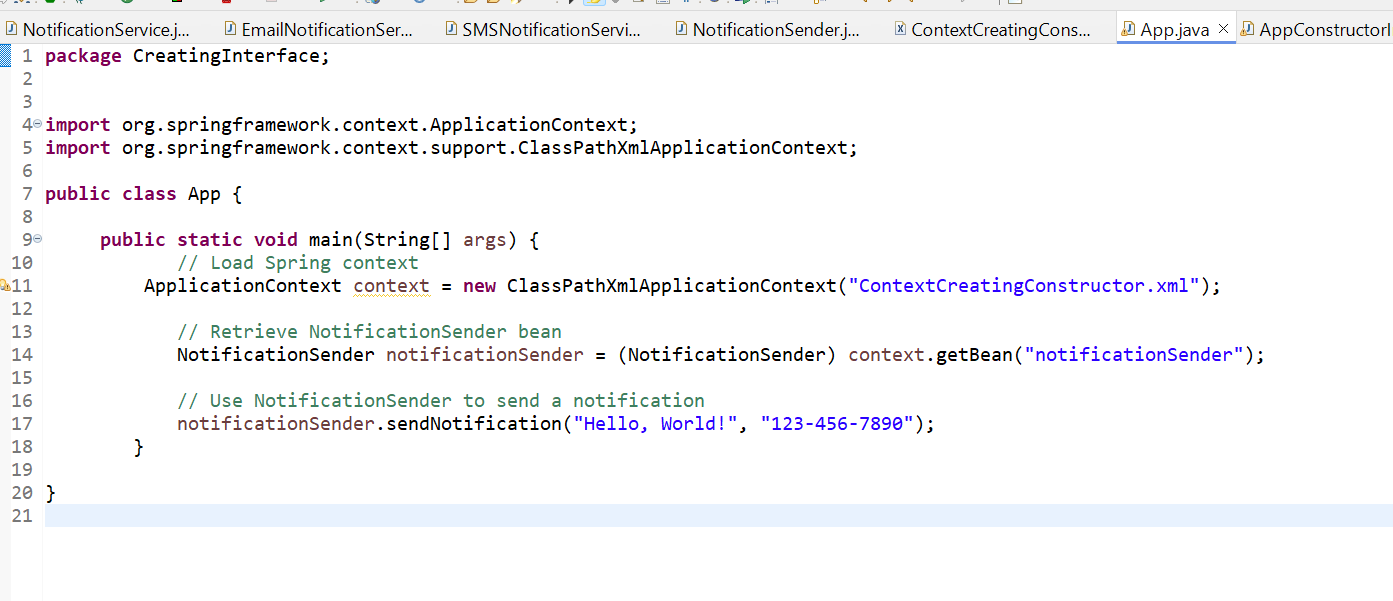
public void sendNotification(String message, String recipient) {

notificationService.sendNotification(message, recipient);

}

}

By depending on the *NotificationService* interface, the *NotificationSender* class doesn't need to know which specific implementation it's using. It only requires an object that adheres to the *NotificationService* contract. This allows for greater flexibility and extensibility. Now, using Spring's dependency injection, you can configure the appropriate implementation to be injected into the *NotificationSender* class based on your needs.



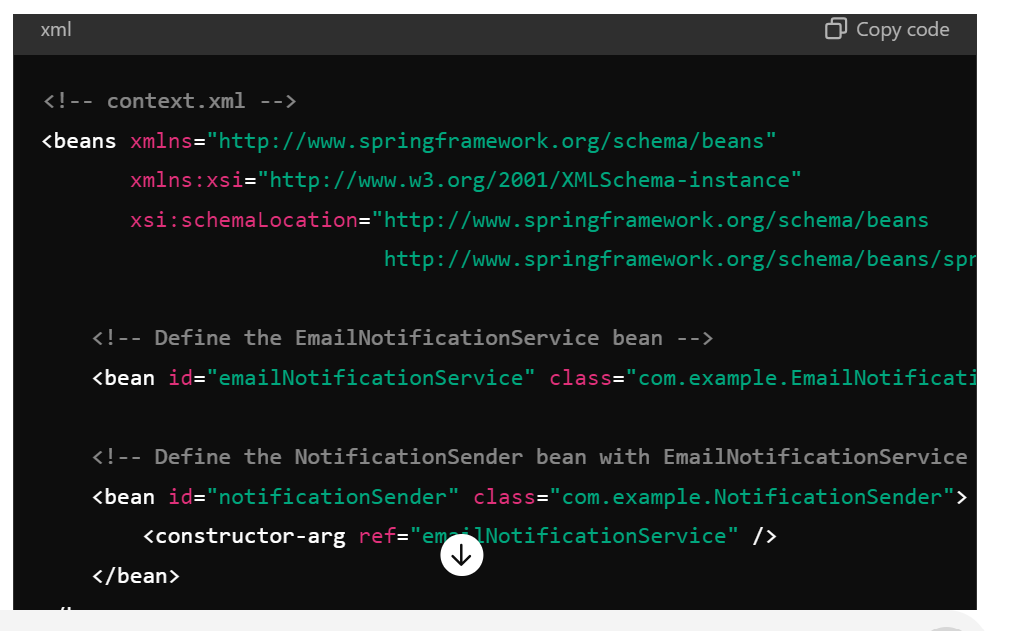
Context.xml path

In your application, you can now retrieve the NotificationSender bean from the Spring context and use it to send notifications. The NotificationSender will use the SMSNotificationService as configured.

****

**XML Configuration:** Define beans and use <constructor-arg> to inject the specific implementation of NotificationService (e.g., SMSNotificationService) into NotificationSender.

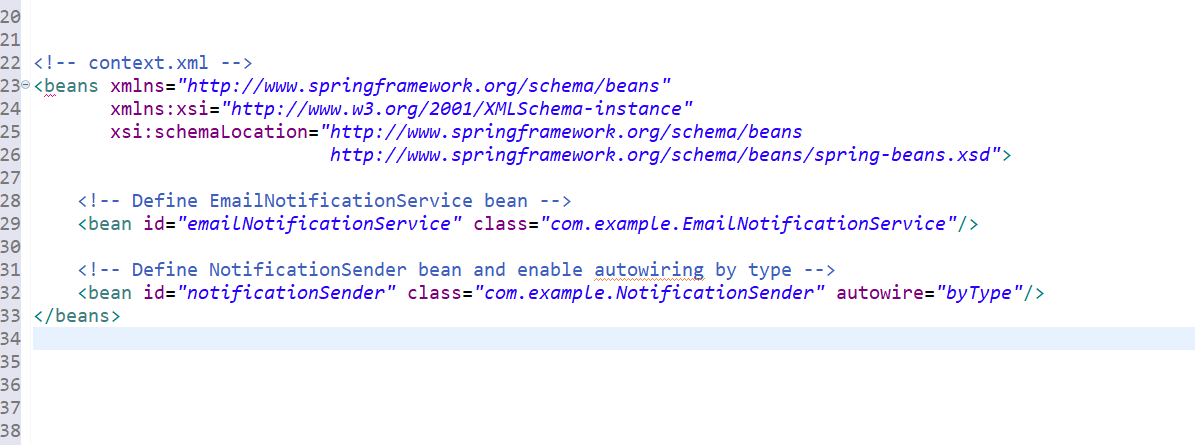
If we want to pass Email service, we have to modify our xml file**.**

****

**Types of Autowiring in Spring XML Configuration**

* 1. **Autowiring by Type**

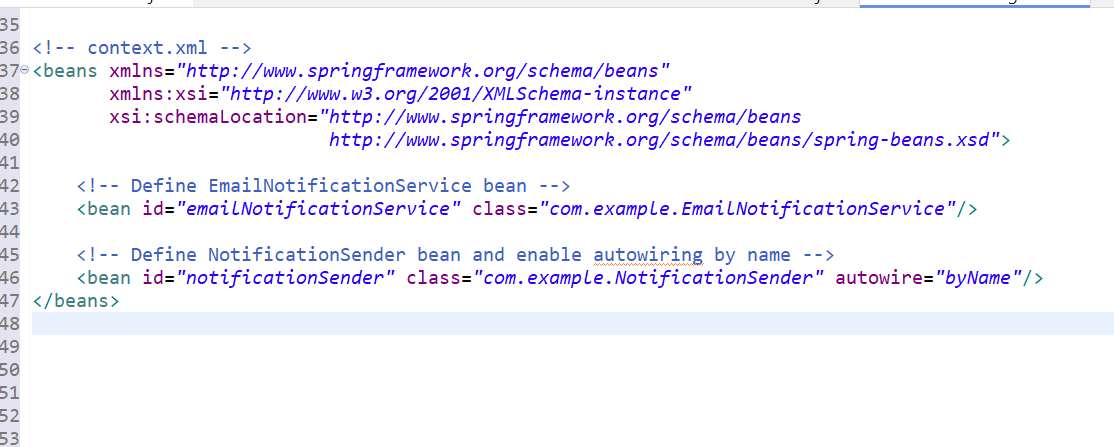
When you use autowiring by type, Spring will automatically inject a bean of the matching type into the property of another bean. This is done using the **autowire="byType"** attribute in the XML configuration**.**

****

In this example, **NotificationSender** will have its **NotificationService** property automatically injected with an instance of **EmailNotificationService** because the type matches.

* 1. **Autowiring by Name**

With autowiring by name, Spring will inject a bean into a property of another bean if the name of the property matches the name of the bean. This is done using the **autowire="byName"** attribute.

****

In this example, if **NotificationSender** has a property named **emailNotificationService**, Spring will inject the bean with that name into the property.

* 1. **Autowiring by Constructor**

Autowiring by constructor is less common in XML but still possible. It means that Spring will try to inject beans by matching constructor arguments. This can be configured using the **autowire="constructor"** attribute.

****

In this example, **NotificationSender's** constructor arguments will be automatically resolved and injected based on the type and number of constructor parameters.

**Summary of Autowiring Types**

* **byType**: Matches and injects a bean of the required type.
* **byName**: Matches and injects a bean by its name.
* **constructor**: Matches and injects beans into constructor arguments.
* **default**: If not specified, Spring defaults to no autowiring, requiring explicit configuration.

**Primary Bean**

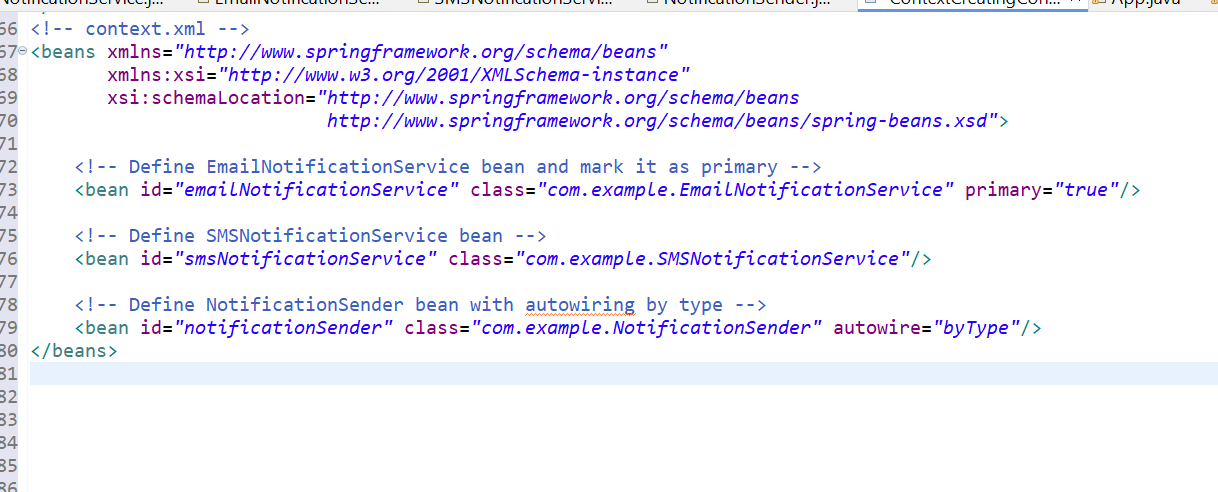
In Spring XML configuration, the concept of a "primary bean" is used to specify which bean should be used when multiple beans of the same type are available. This is particularly useful for resolving ambiguities in dependency injection when there are multiple beans of the same type but only one should be injected by default.

**Example of Primary Bean Configuration**

In XML configuration, you specify a primary bean using the primary attribute of the bean element. This attribute is available starting from Spring 3.0.

Let’s say you have multiple beans of type NotificationService and you want to indicate which one should be used by default**:**

**XML Configuration:**

****

**Explanation**

* **Primary Bean (primary="true"):** The **emailNotificationService** bean is marked as primary. This means that if a bean of type **NotificationService** is required for autowiring and no specific bean is qualified, Spring will use **emailNotificationService** by default**.**
* **Other Beans: smsNotificationService** is another bean of the same type but is not marked as primary.
* **Autowiring:** In the **NotificationSender** bean, if autowiring by type is used **(autowire="byType"),** Spring will inject emailNotificationService into the **notificationService** property of N**otificationSender** because it is marked as primary**.**

**Why Use a Primary Bean?**

* **Resolve Ambiguities:** When multiple beans of the same type are present, marking one as primary helps Spring resolve which bean to inject by default.
* **Simplify Configuration:** It reduces the need to use qualifiers or specific configuration to select a default bean for injection**.**

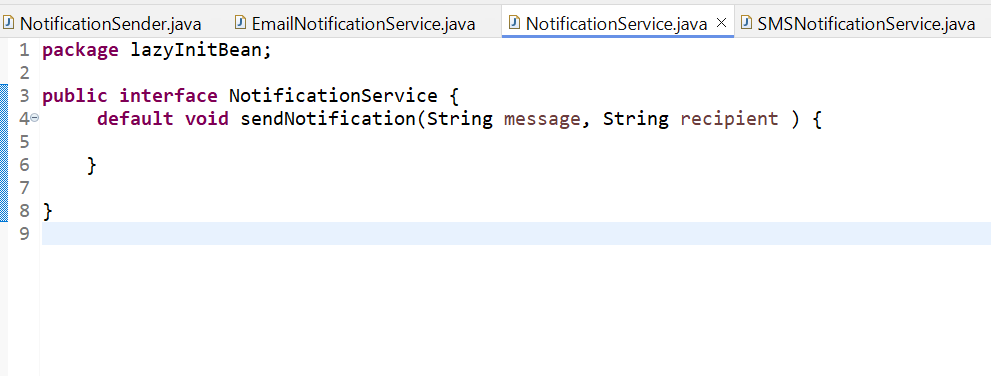
**Lazyinit bean**

In Spring, a bean is instantiated and initialized as soon as the application context is created by default. However, in some cases, you might want to delay the creation of a bean until it is actually needed. This is where "lazy initialization" comes into play.

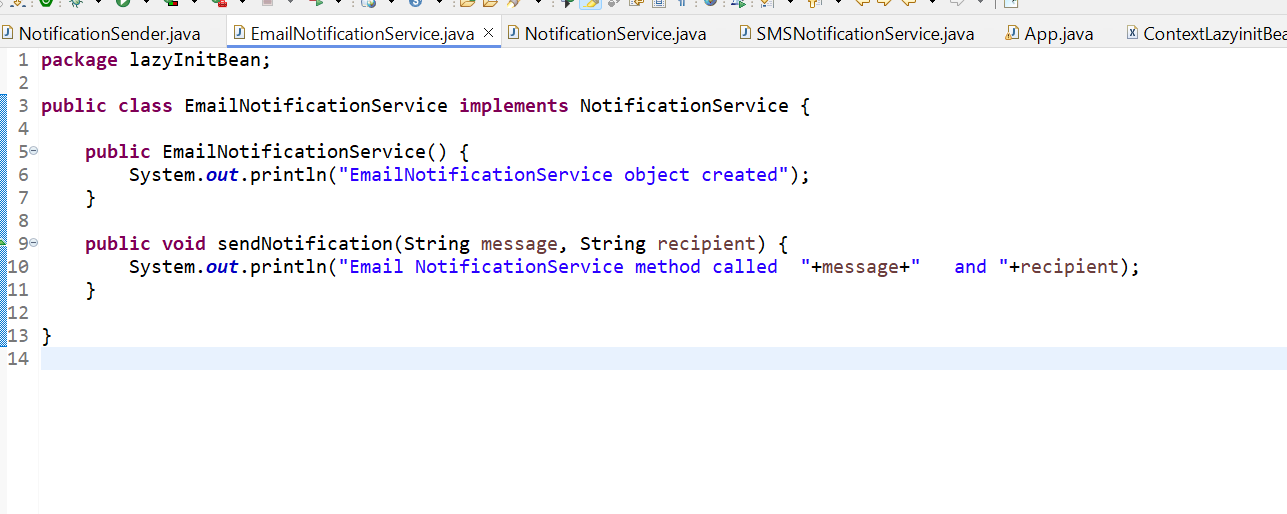
**What is Lazy Initialization?**

Lazy initialization refers to the practice of delaying the creation and initialization of a bean until it is first requested. This can be beneficial for improving startup performance or reducing resource consumption, especially if certain beans are not always used.

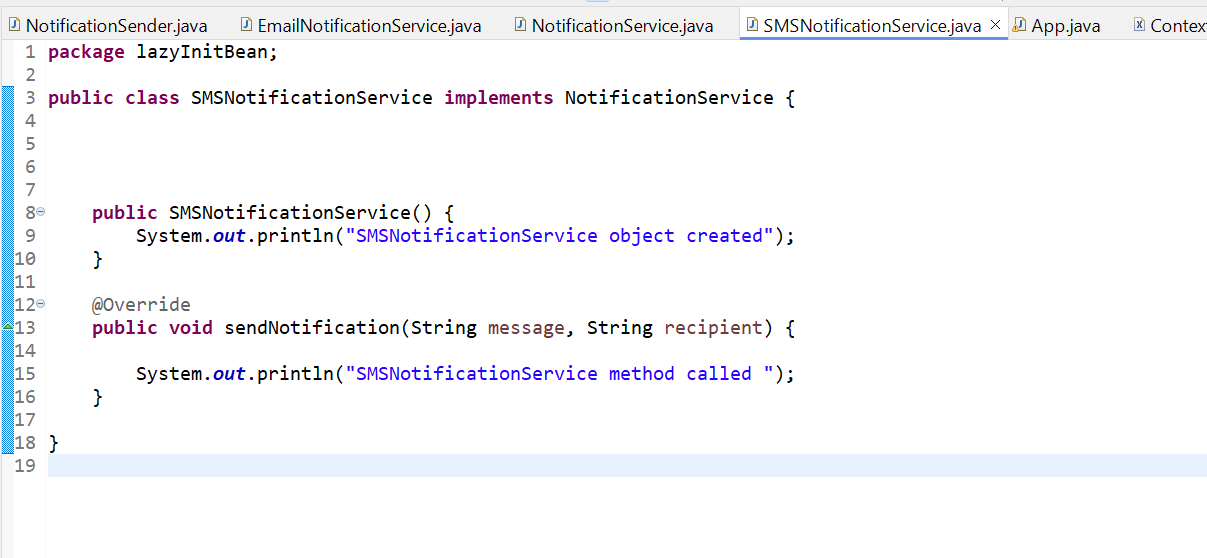
**NotificationService.java**

****

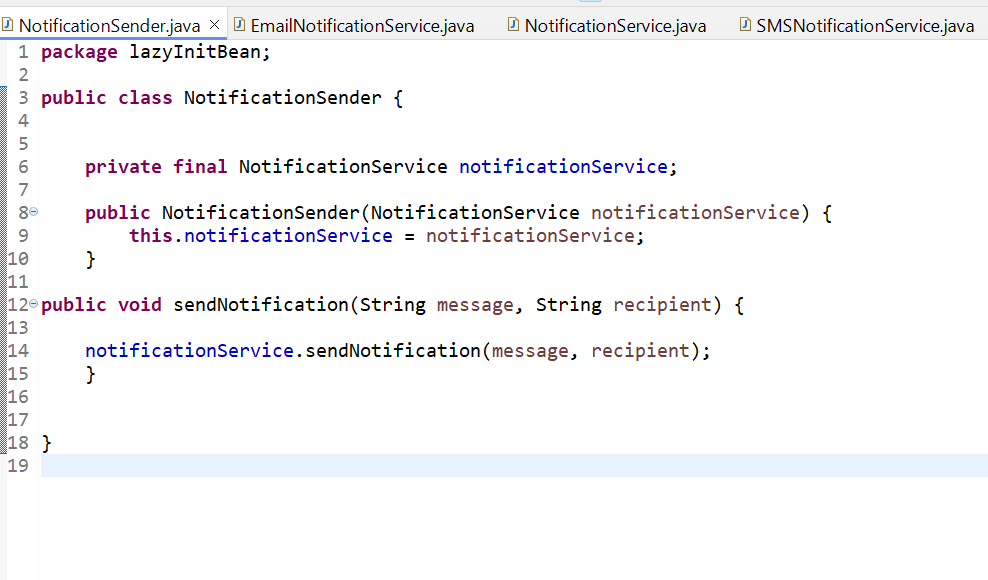
**EmailNotificationService.java**

****

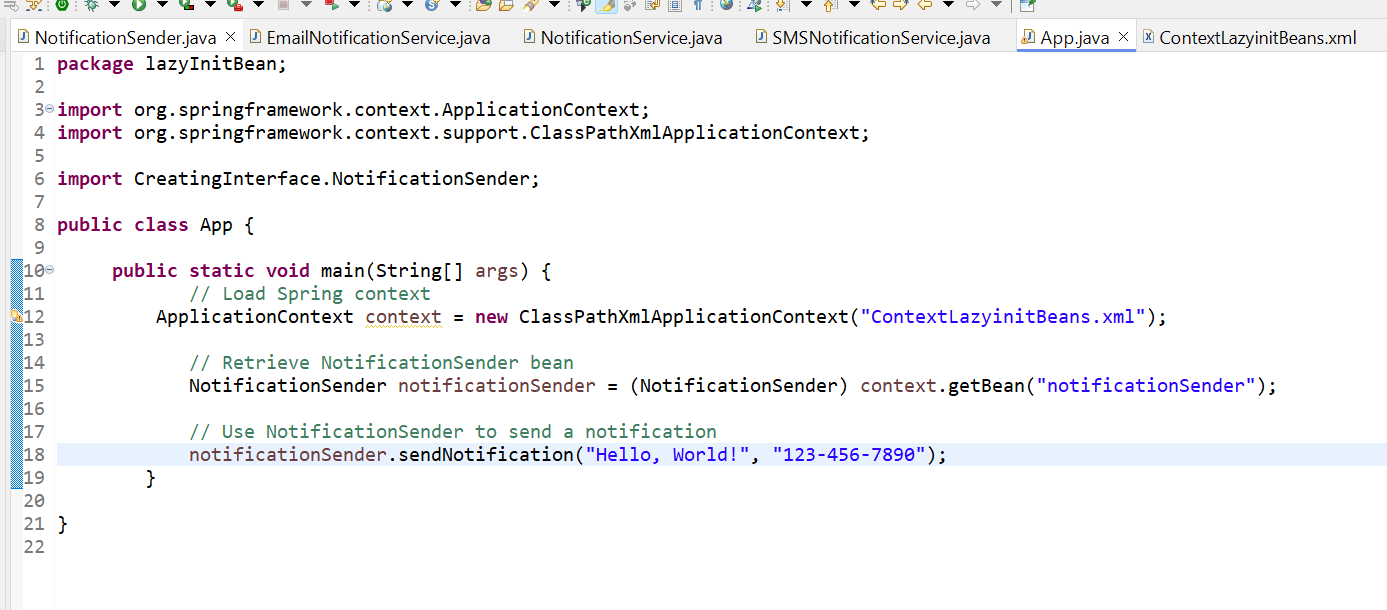
**SMSNotificationService.java**

****

**NotificationSender.java**

****

**App.java**

****

**Expected Output**

1. When App is run, **SMSNotificationService** will be initialized immediately because it is eagerly initialized by default.
2. **EmailNotificationService** will be initialized only when it is first needed. Since **notificationSender** depends on **emailNotificationService**, it will be initialized when **notificationSender** is accessed.

**Output:**

SMSNotificationService initialized

EmailNotificationService initialized

Sending email to example@example.com with message: Hello, World!

**Summary**

* Eager Initialization: By default, SMSNotificationService is eagerly initialized as it is not marked with lazy-init="true".
* Lazy Initialization: EmailNotificationService is lazily initialized and only created when notificationSender is retrieved from the application context.

In Spring, "getting a bean by type" refers to retrieving a bean instance from the Spring application context based on its type rather than its ID. When you use getBean(Class<T> requiredType) method, Spring returns the bean that matches the given type. This approach can be useful in scenarios where you have multiple beans of different types or when you want to avoid hardcoding bean IDs.

**Example of getBean by Type**

Let's use an example based on XML bean configurations to illustrate how this works:

**1. Define Beans in XML Configuration**

Here’s an example of how you might define beans in an XML configuration file (context.xml):

Xml



**2. Define the Bean Classes**

Assuming you have two implementations of NotificationService:

package com.example;

public interface NotificationService {

void sendNotification(String message, String recipient);

}

public class EmailNotificationService implements NotificationService {

@Override

public void sendNotification(String message, String recipient) {

System.out.println("Email Notification: " + message + " to " + recipient);

}

}

public class SMSNotificationService implements NotificationService {

@Override

public void sendNotification(String message, String recipient) {

System.out.println("SMS Notification: " + message + " to " + recipient);

}

}

**3. Get Beans by Type in Code**

Here’s how you can get beans by their type using the ApplicationContext:

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.NotificationService;

public class App {

public static void main(String[] args) {

// Load Spring context from XML configuration

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

// Retrieve beans by type

NotificationService emailService = context.getBean(EmailNotificationService.class);

NotificationService smsService = context.getBean(SMSNotificationService.class);

// Use the beans

emailService.sendNotification("Hello via Email!", "email@example.com");

smsService.sendNotification("Hello via SMS!", "123-456-7890");

}

}

**Key Points to Note**

1. **Bean Definition**:
   * Beans are defined with unique IDs in the XML configuration file.
   * Each bean is associated with a specific class that implements an interface.
2. **Bean Retrieval**:
   * You use context.getBean(Class<T> requiredType) to retrieve a bean by its type.
   * Spring will return the instance of the bean that matches the specified type.
3. **Type Matching**:
   * If multiple beans implement the same interface, you can still retrieve a specific bean by type if you know the exact class.
   * If multiple beans of the same type exist, Spring will throw an exception if it cannot resolve the ambiguity. You may need to use qualifiers or other methods to disambiguate.