# Dependency Injection

Introduction

## Dependency Injection (DI)

- Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.
- Dependency Injection generally means passing a dependent object as a parameter to a method, rather than having the method create the dependent object.
- Dependency Injection makes our programming code loosely coupled.

## Dependency Injection

Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

## Spring Container

Object Factory

- Primary functions
  - Create and manage objects *(Inversion of Control)*
  - Inject object's dependencies *(Dependency Injection)*
- Configuring Spring Container
  - XML configuration file (legacy)
  - Java Annotation (Modern)
  - Java Source Code (Modern)

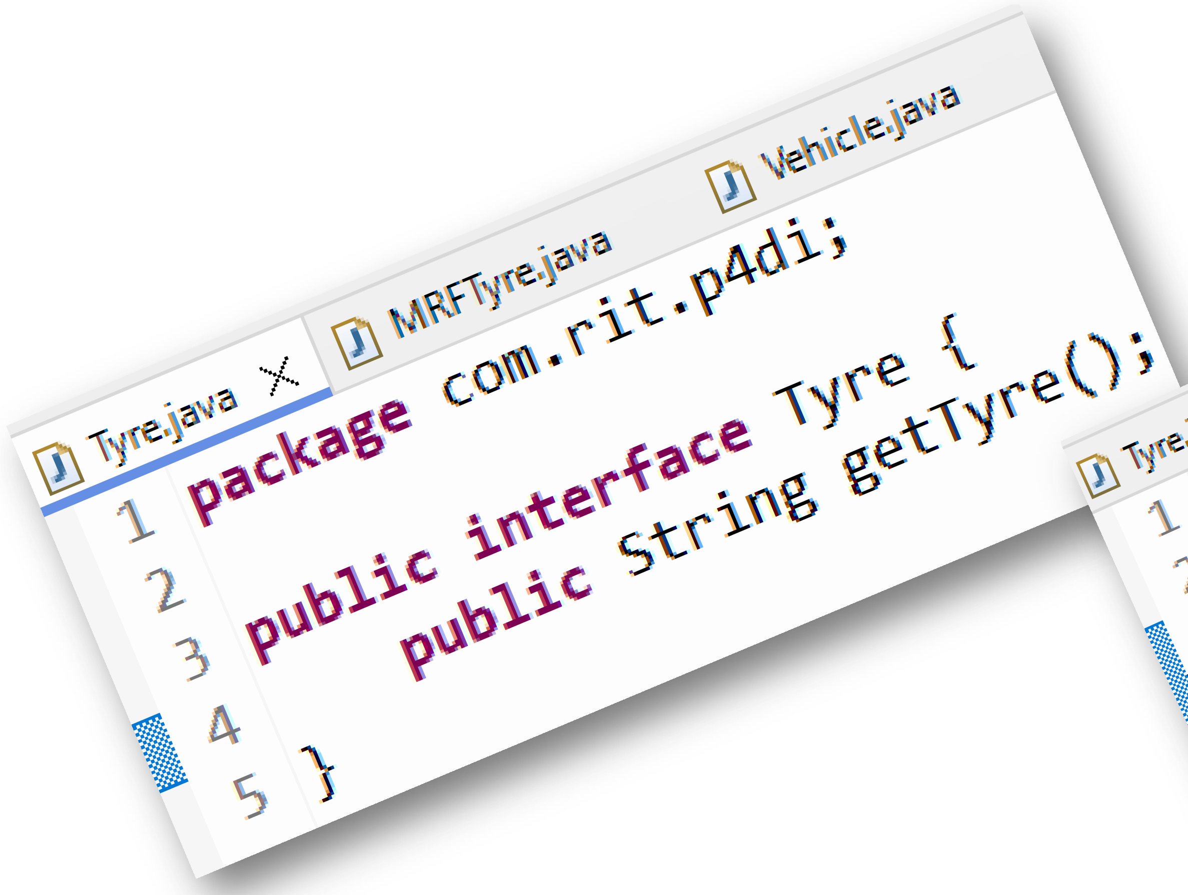
## Spring Development Process

1. Defining the dependency.
   a) Create an interface “TyreService”
   b) Create a class “MRFTyreService” implements “TyreService”
2. Injecting the dependency
   a) Add a new method in the “Vehicle” interface to getTyre
   b) Create a constructor
   c) Create a private dependency member “TyreService”
   d) Inject it with the constructor
   e) Implement the new method with injected dependency

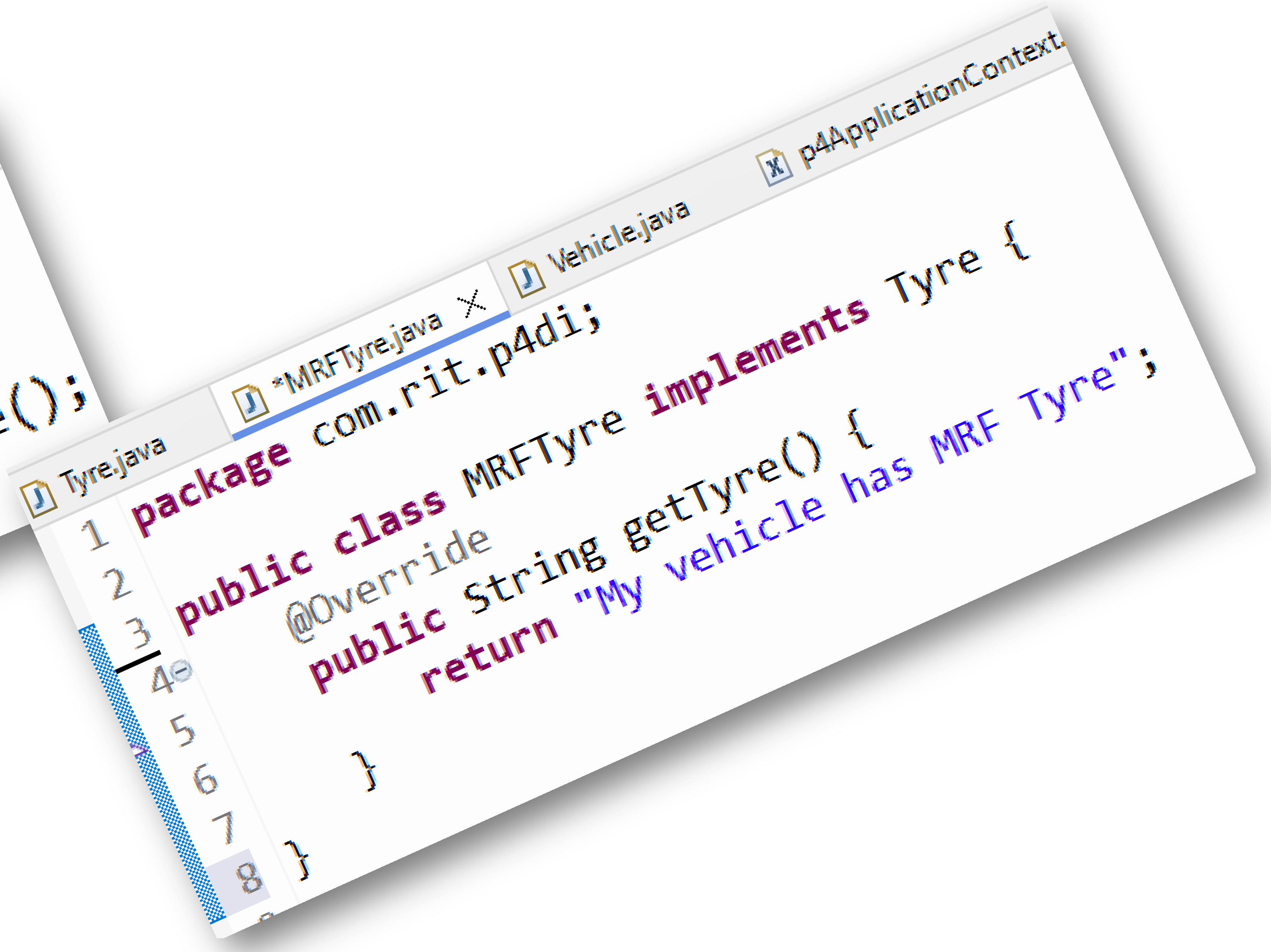
## Spring Development Process

1. Configure the dependency.
    a) Define a bean for dependency class “MRFTyreService”
    b) Add dependency bean as a constructor-arg
2. Invoke the dependency method in Main class
    a) Invoke the method

## Create Tyre interface and a class

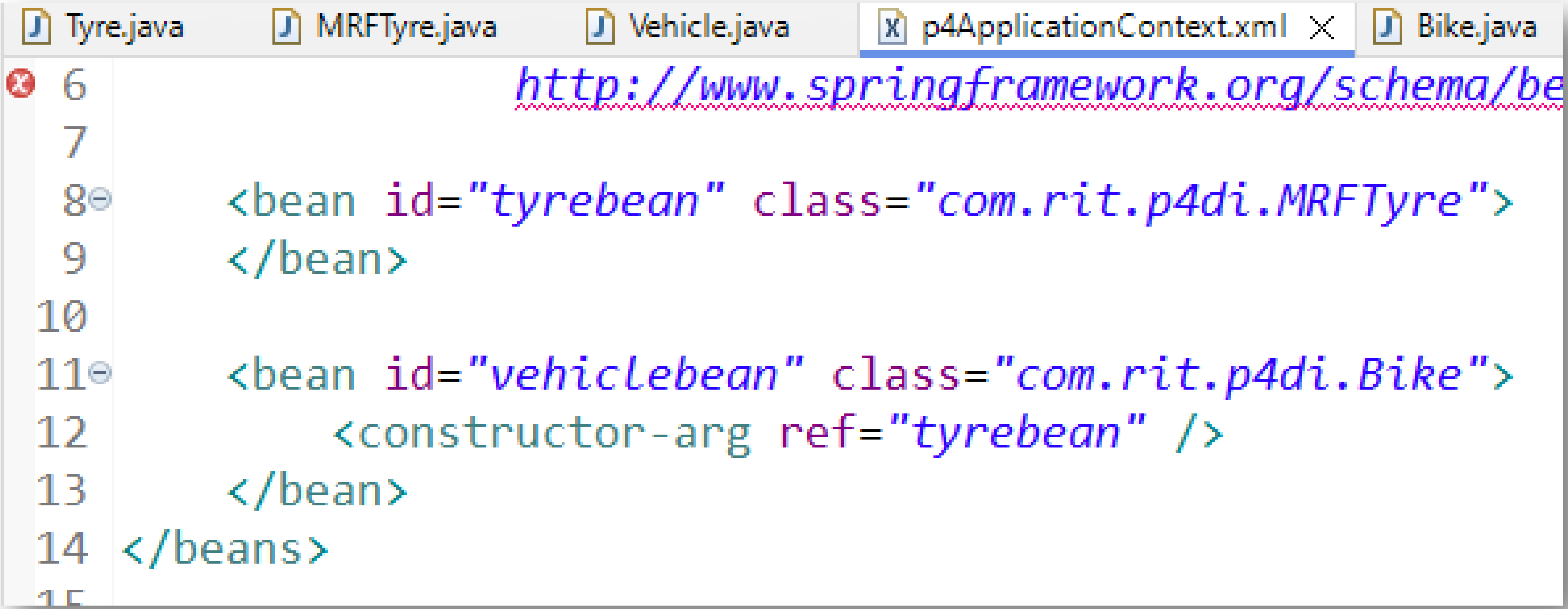


```java
package com.rit.p4di;
public interface Tyre {
    public String getTyre();

}
```



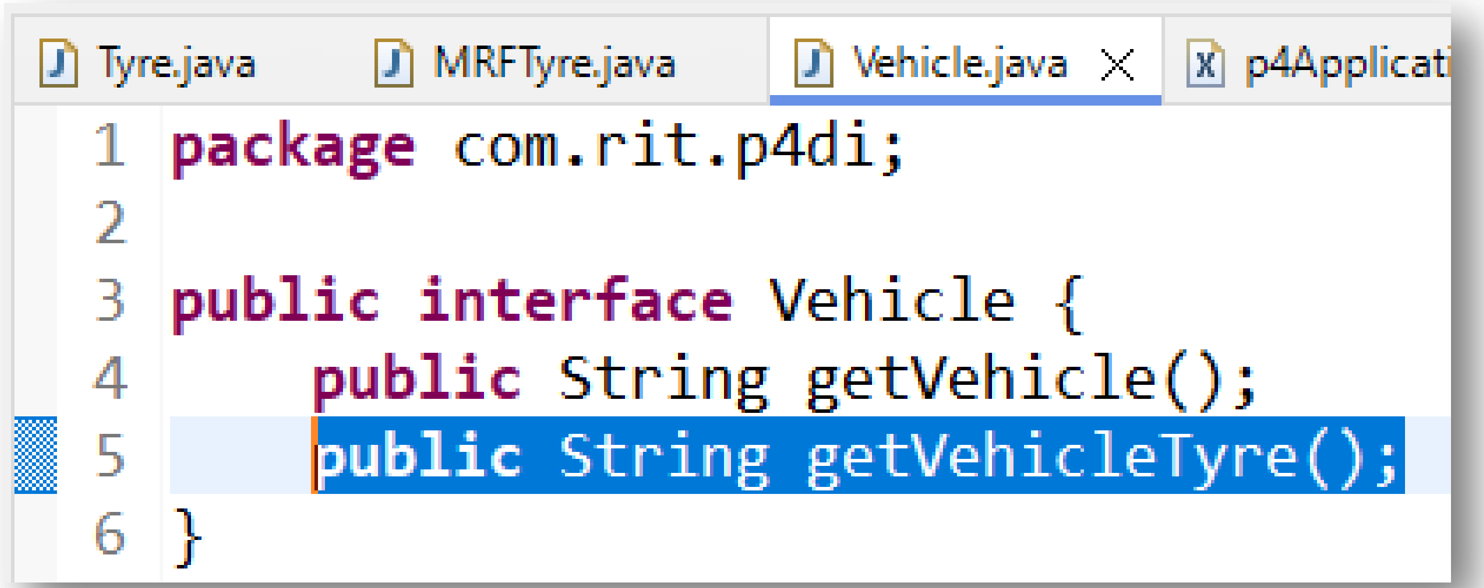
```java
package com.rit.p4di;
public class MRFTyre implements Tyre {
    @Override
    public String getTyre() {
        return "My vehicle has MRF Tyre";

    }

}
```

## Create Tyre object
## Inject it with vehicle using constructor



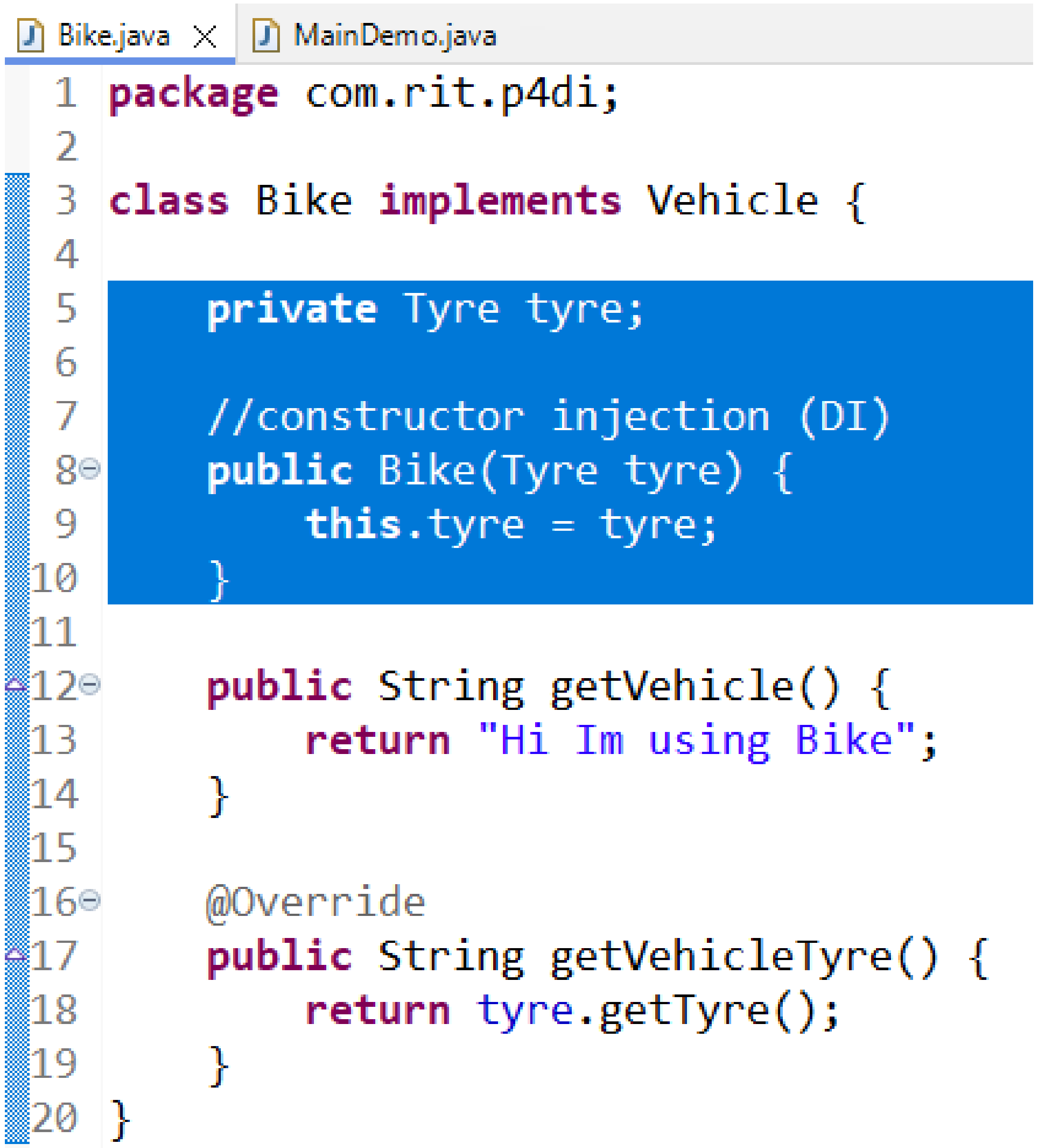
```
            http://www.springframework.org/schema/be

    <bean id="tyrebean" class="com.rit.p4di.MRFTyre">
    </bean>

    <bean id="vehiclebean" class="com.rit.p4di.Bike">
        <constructor-arg ref="tyrebean" />
    </bean>
</beans>
```

# Declare a method to display tyre details



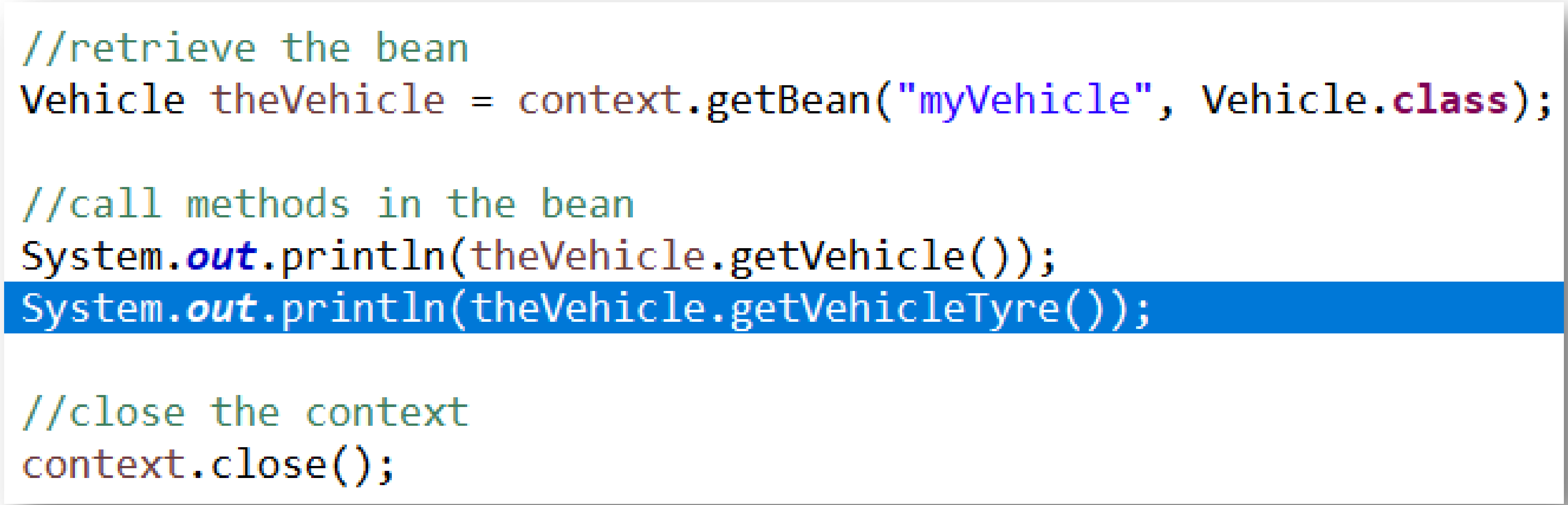
```java
package com.rit.p4di;

public interface Vehicle {
	public String getVehicle();
	public String getVehicleTyre();
}
```

Get the constructor injected tyre and Display the tyre details



```
package com.rit.p4di;

class Bike implements Vehicle {

    private Tyre tyre;

    //constructor injection (DI)
    public Bike(Tyre tyre) {
        this.tyre = tyre;
    }

    public String getVehicle() {
        return "Hi Im using Bike";
    }

    @Override
    public String getVehicleTyre() {
        return tyre.getTyre();
    }
}
```

## Display the tyre along with vehicle details



```
//retrieve the bean
Vehicle theVehicle = context.getBean("myVehicle", Vehicle.class);

//call methods in the bean
System.out.println(theVehicle.getVehicle());
System.out.println(theVehicle.getVehicleTyre());

//close the context
context.close();
```

# Dependency Injection

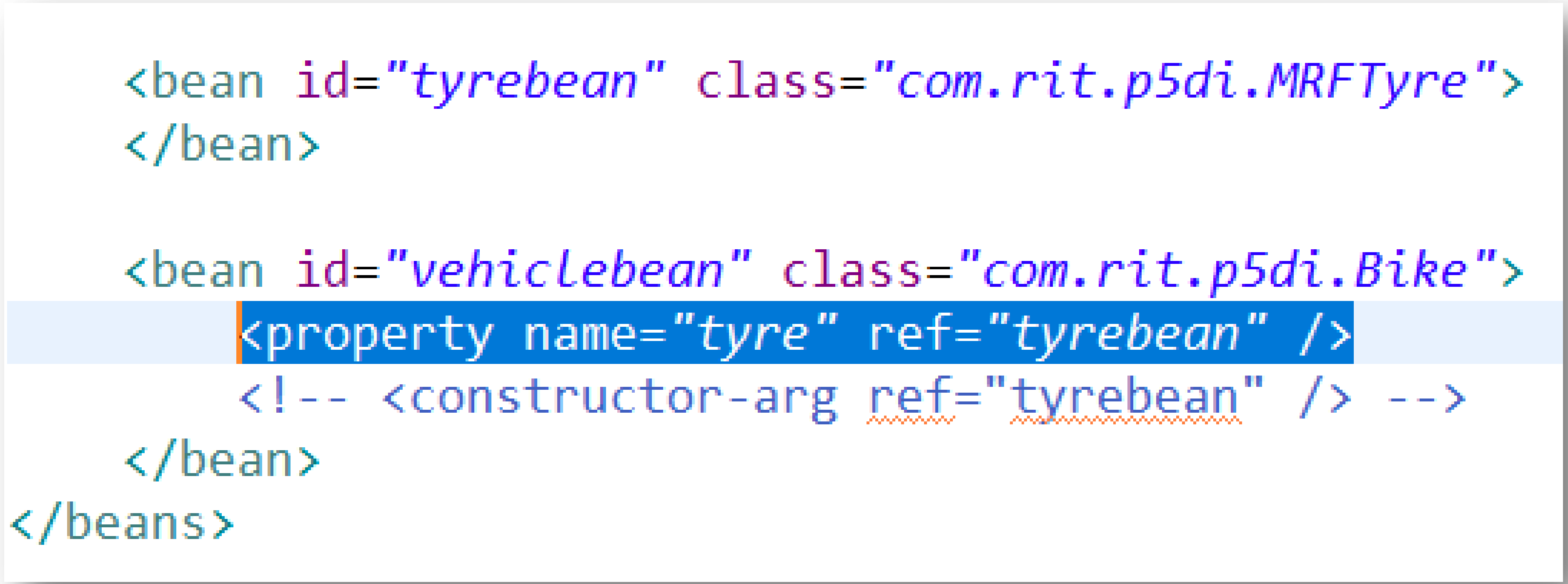
Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

## Spring Development Process

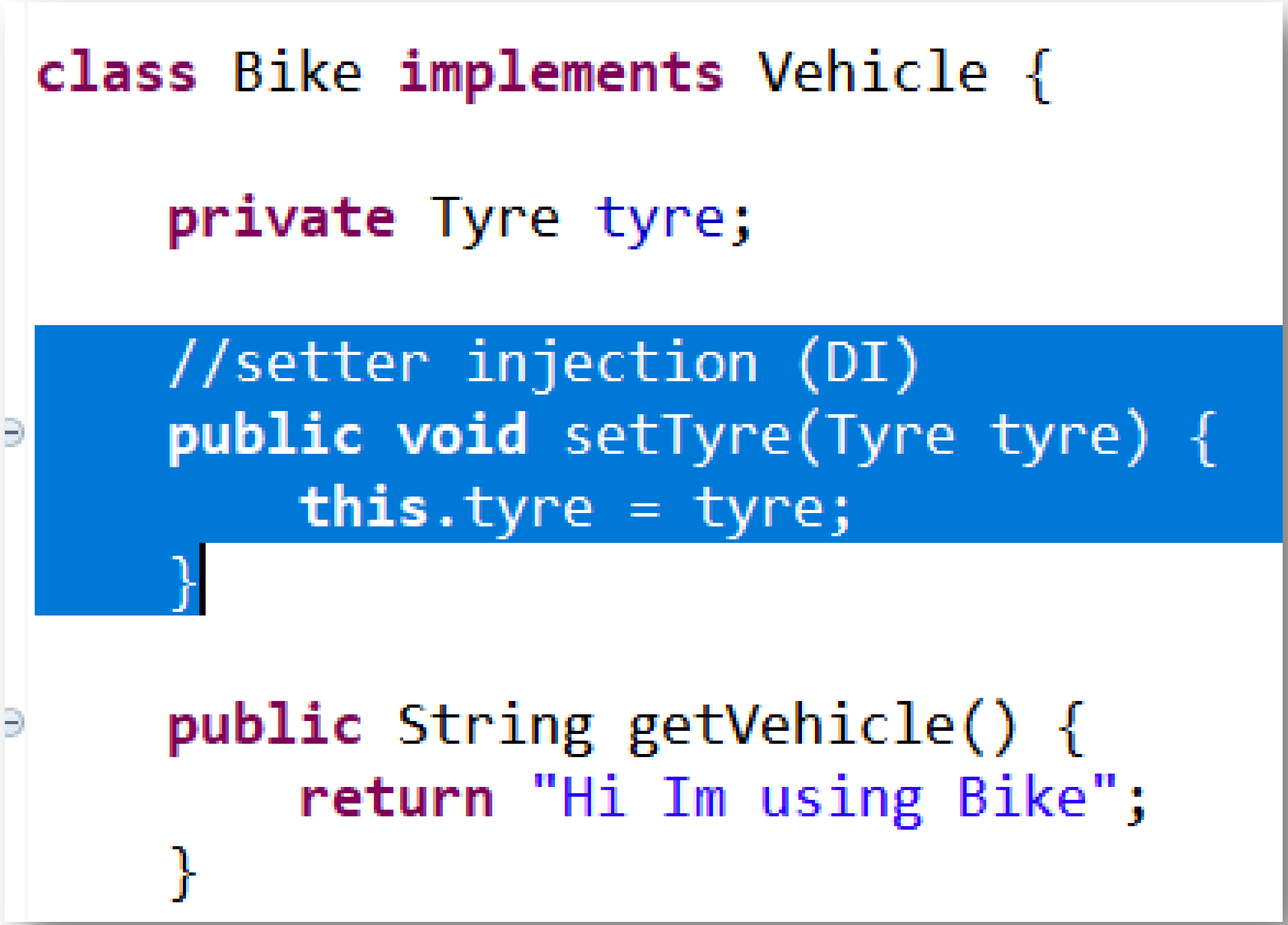
1. In xml file replace constructor-arg with property
2. In Bike class replace the constructor with setter method

## Replace constructor-arg with property



```xml
    <bean id="tyrebean" class="com.rit.p5di.MRFTyre">
    </bean>

    <bean id="vehiclebean" class="com.rit.p5di.Bike">
        <property name="tyre" ref="tyrebean" />
        <!-- <constructor-arg ref="tyrebean" /> -->
    </bean>
</beans>
```

## Replace the constructor with setter method



```
class Bike implements Vehicle {

    private Tyre tyre;

    //setter injection (DI)
    public void setTyre(Tyre tyre) {
        this.tyre = tyre;
    }

    public String getVehicle() {
        return "Hi Im using Bike";
    }
```

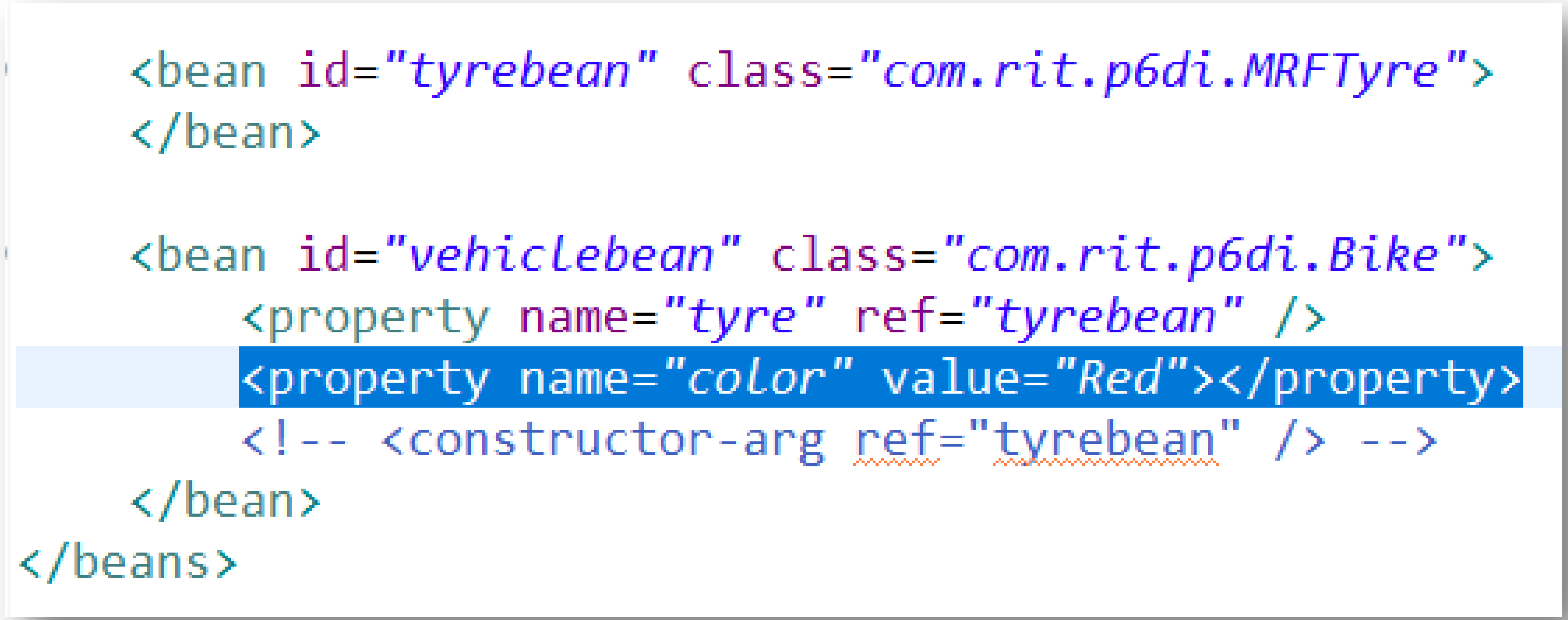
## Injecting String literal

## Injecting String Literal

- Inject values through property tag in applicationContext.xml

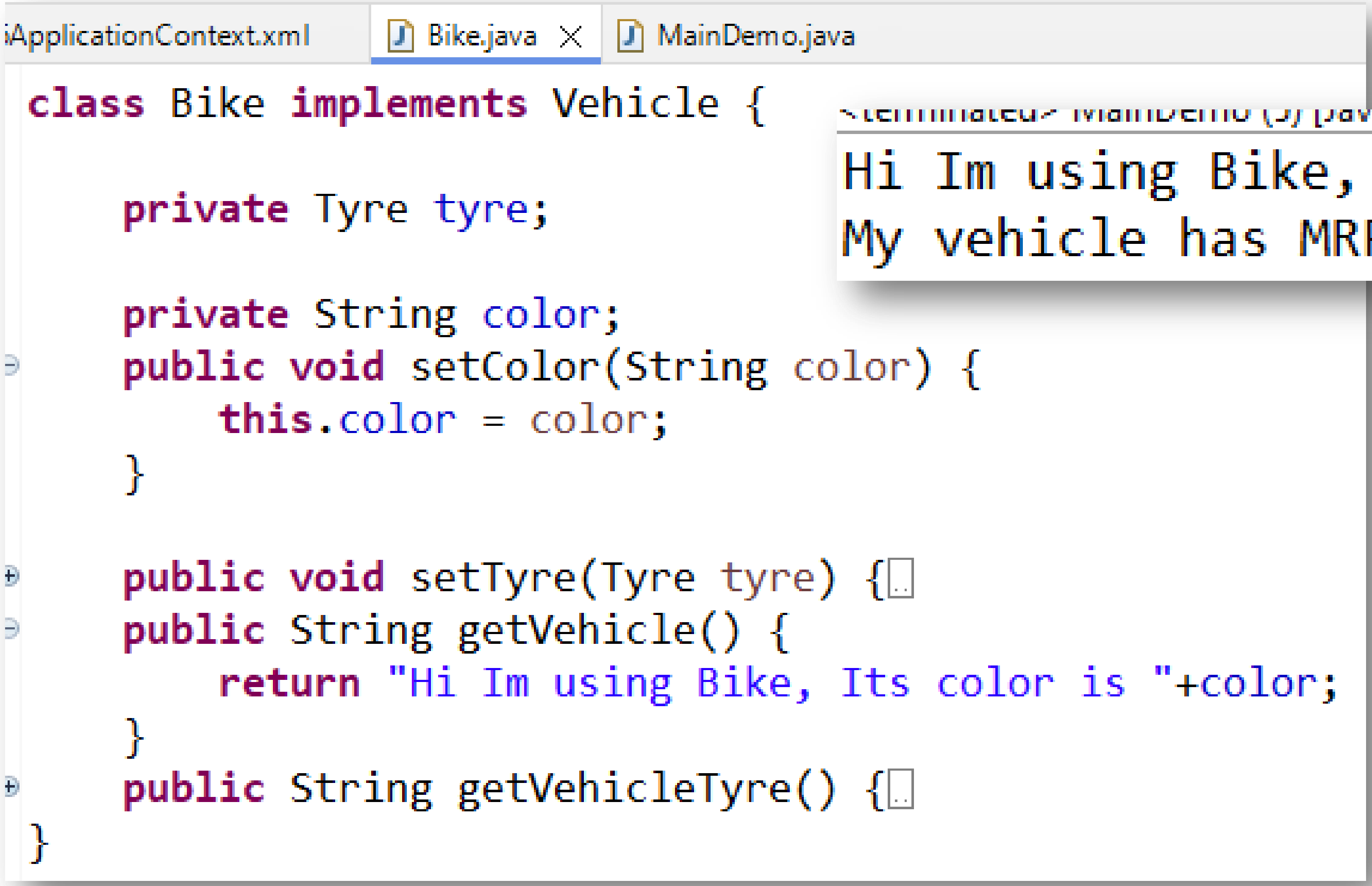
- Add private property in Bike class
- Generate setter method.
- Use the property value in a method

# Injecting string value instead of object reference

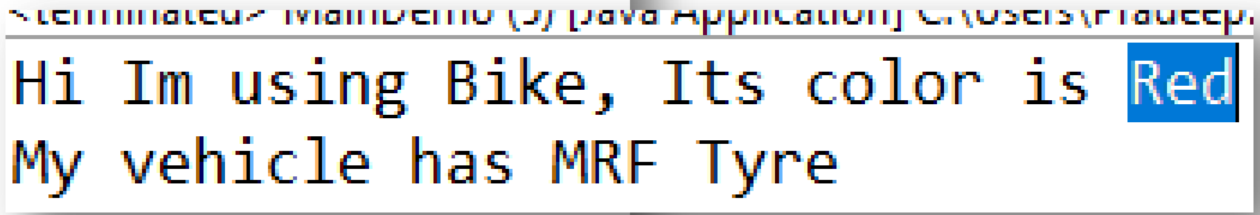


```
    <bean id="tyrebean" class="com.rit.p6di.MRFTyre">
    </bean>

    <bean id="vehiclebean" class="com.rit.p6di.Bike">
        <property name="tyre" ref="tyrebean" />
        <property name="color" value="Red"></property>
        <!-- <constructor-arg ref="tyrebean" /> -->
    </bean>
</beans>
```

Add a field with setter to get the injected string value



```
class Bike implements Vehicle {

    private Tyre tyre;

    private String color;
    public void setColor(String color) {
        this.color = color;
    }

    public void setTyre(Tyre tyre) {
    public String getVehicle() {
        return "Hi Im using Bike, Its color is "+color;
    }
    public String getVehicleTyre() {
}
```



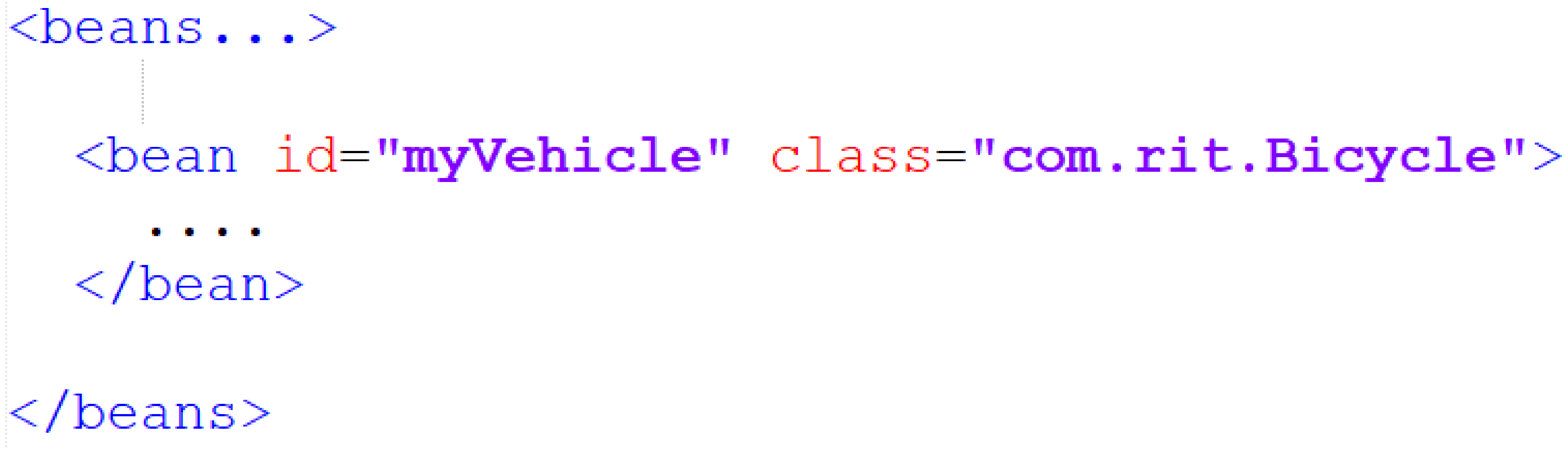
```
Hi Im using Bike, Its color is Red
My vehicle has MRF Tyre
```

## Scope

## Bean Scopes

- Scope refer to the lifecycle of a bean
- How long does the bean live
- How many instances are created
- How is the bean shared

## Default Scope: Singleton



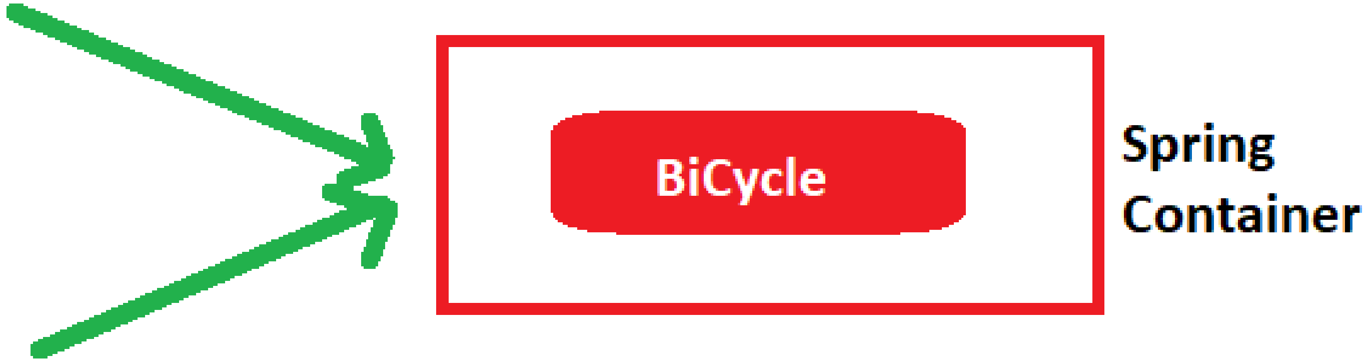
```
<beans...>

  <bean id="myVehicle" class="com.rit.Bicycle">
    ....
  </bean>

</beans>
```

## What is a Singleton?

- Spring container creates only one instance of the bean, by default
- It is cached in memory
- All requests of the bean, will return a shared reference to the same bean.

## What is a Singleton?

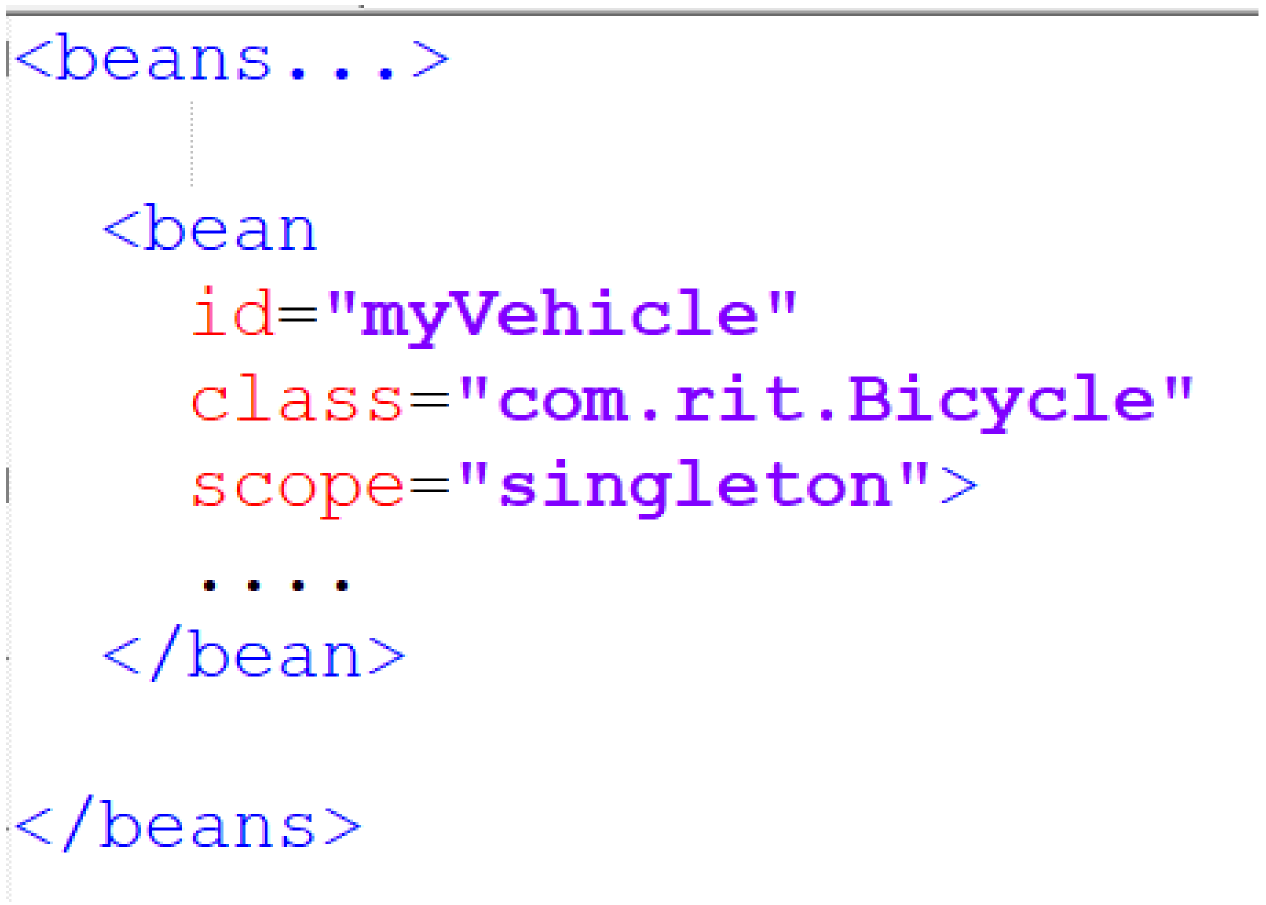
```
Vehicle v1 = context.getBean("myVehicle",Vehicle.class);
```



```
Vehicle v2 = context.getBean("myVehicle",Vehicle.class);
```

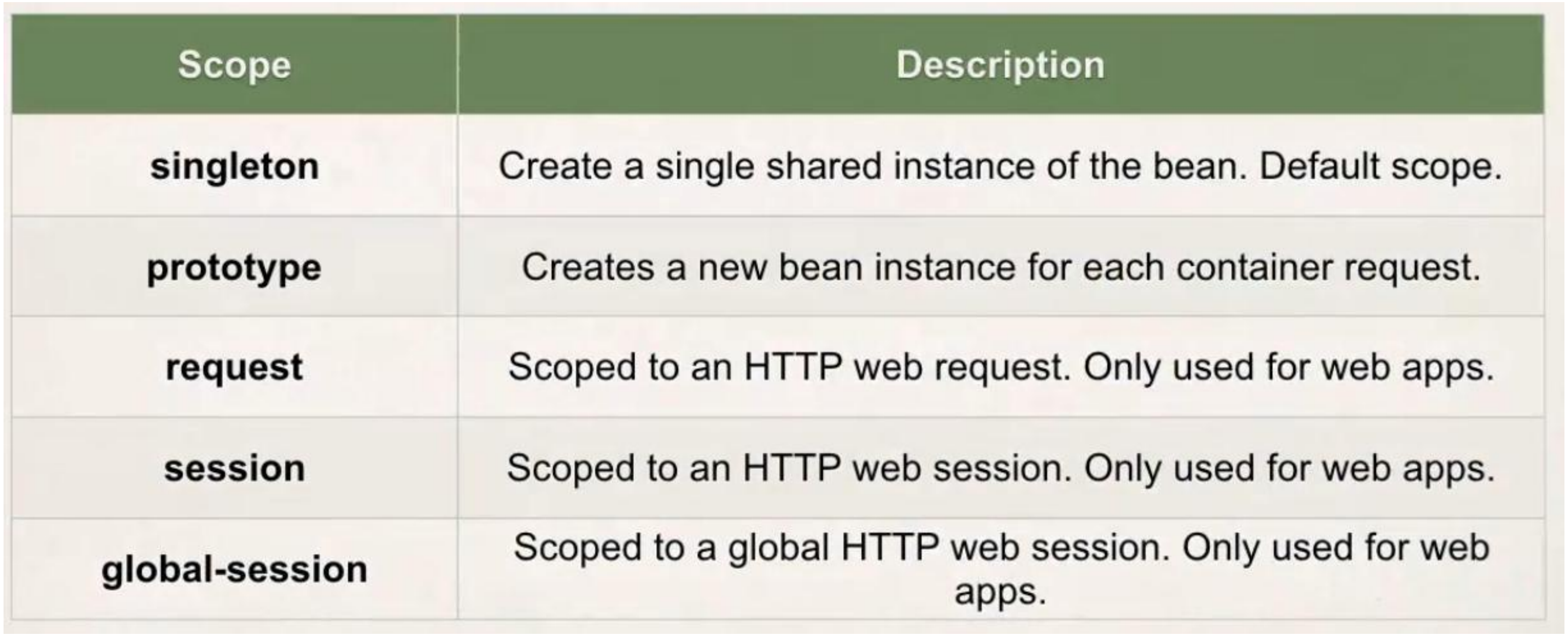
**Singleton is best suited for stateless bean**

# Explicitly specify bean scope



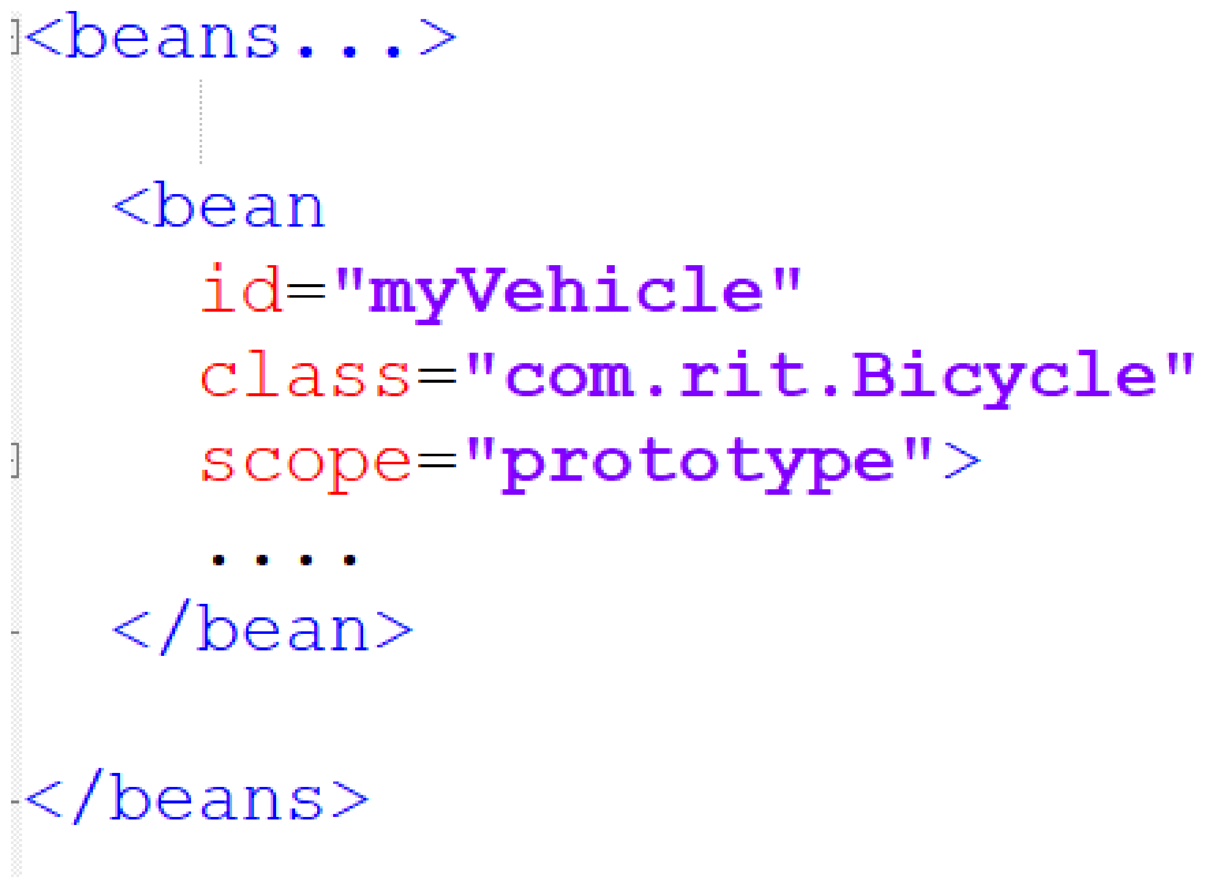
```
<beans...>

  <bean
    id="myVehicle"
    class="com.rit.Bicycle"
    scope="singleton">
    ....
  </bean>

</beans>
```

## Additional Spring bean scopes



| Scope | Description |
|---|---|
| **singleton** | Create a single shared instance of the bean. Default scope. |
| **prototype** | Creates a new bean instance for each container request. |
| **request** | Scoped to an HTTP web request. Only used for web apps. |
| **session** | Scoped to an HTTP web session. Only used for web apps. |
| **global-session** | Scoped to a global HTTP web session. Only used for web apps. |

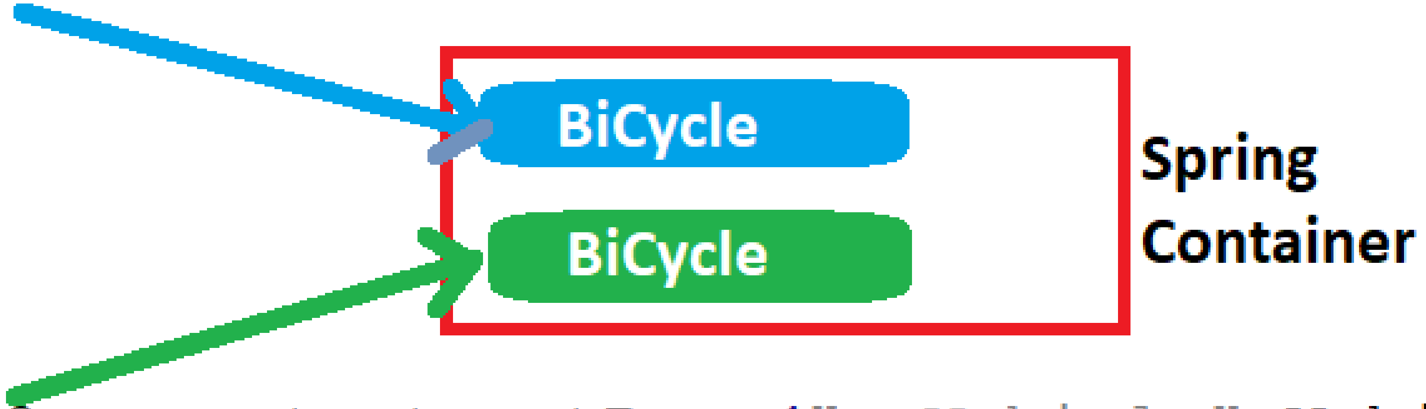
## Prototype scope: new object for every request



```
<beans...>


  <bean
     id="myVehicle"
     class="com.rit.Bicycle"
     scope="prototype">
     ....
  </bean>

</beans>
```

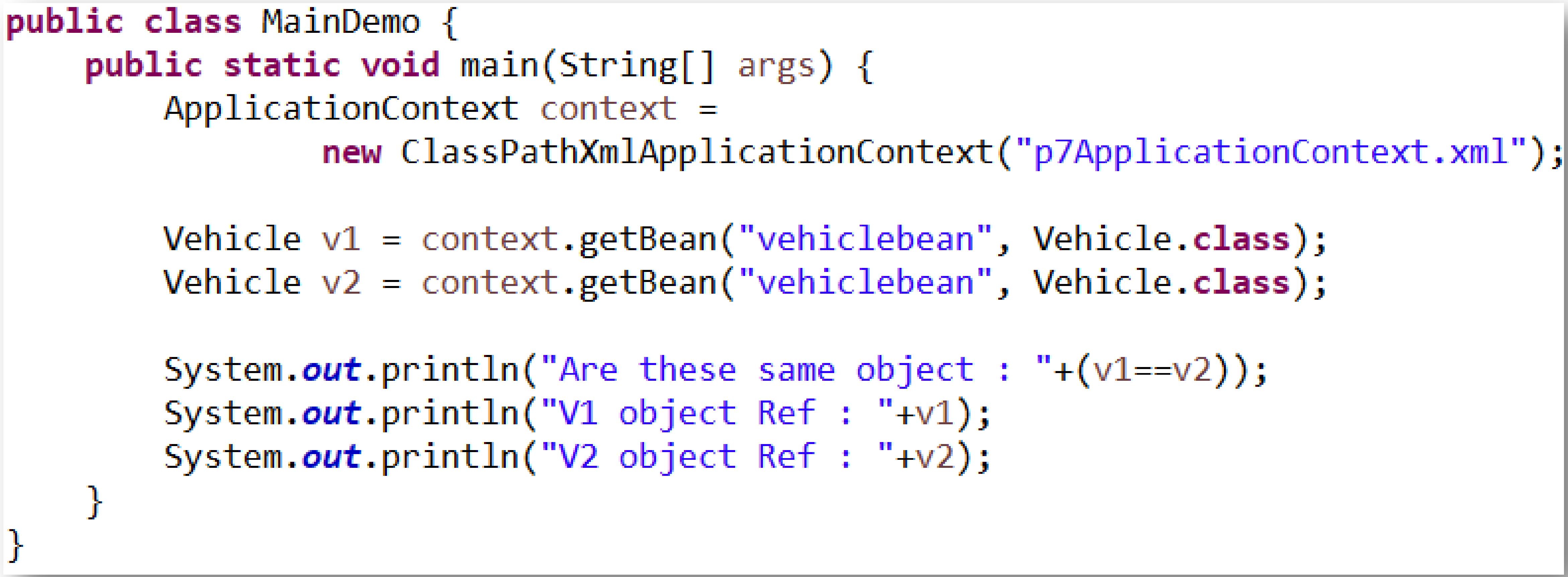
## Prototype scope

```
Vehicle v1 = context.getBean("myVehicle",Vehicle.class);
```



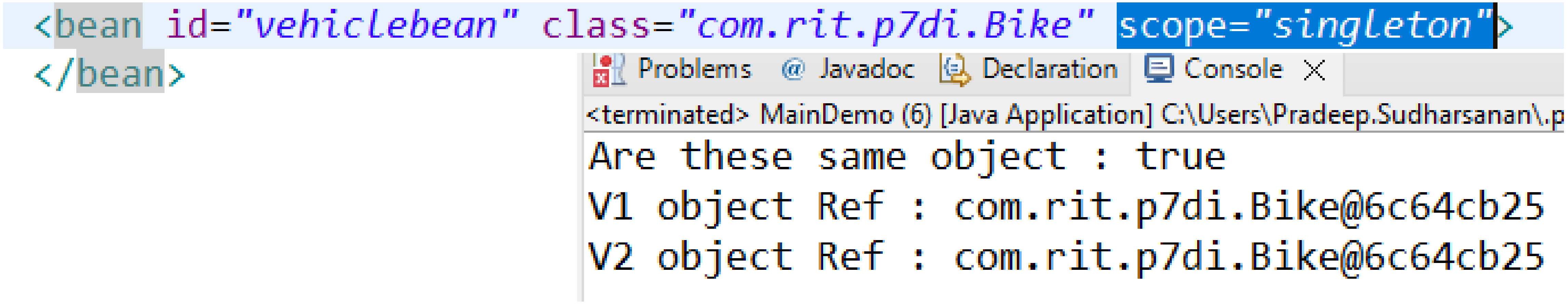
```
Vehicle v2 = context.getBean("myVehicle",Vehicle.class);
```

# Main Class



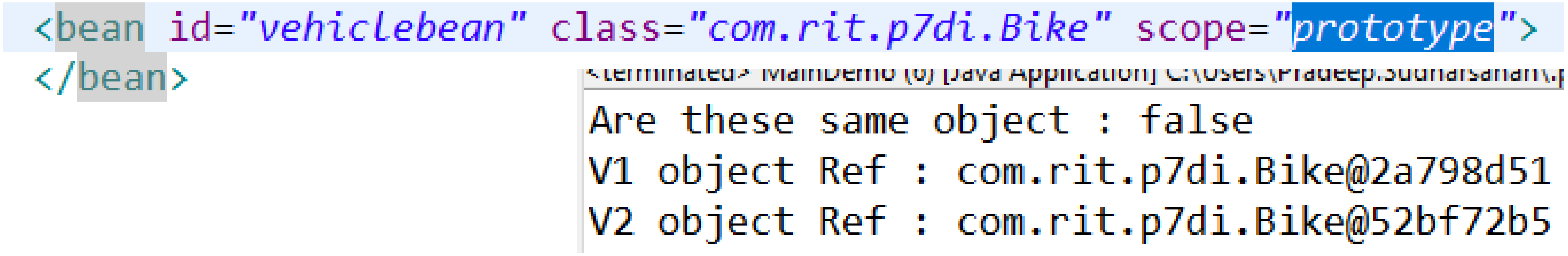
```
public class MainDemo {
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("p7ApplicationContext.xml");

        Vehicle v1 = context.getBean("vehiclebean", Vehicle.class);
        Vehicle v2 = context.getBean("vehiclebean", Vehicle.class);

        System.out.println("Are these same object : "+(v1==v2));
        System.out.println("V1 object Ref : "+v1);
        System.out.println("V2 object Ref : "+v2);
    }
}
```

# Add Scope in context file



```xml
<bean id="vehiclebean" class="com.rit.p7di.Bike" scope="singleton">
</bean>
```

```
<terminated> MainDemo (6) [Java Application] C:\Users\Pradeep.Sudharsanan\.p
Are these same object : true
V1 object Ref : com.rit.p7di.Bike@6c64cb25
V2 object Ref : com.rit.p7di.Bike@6c64cb25
```

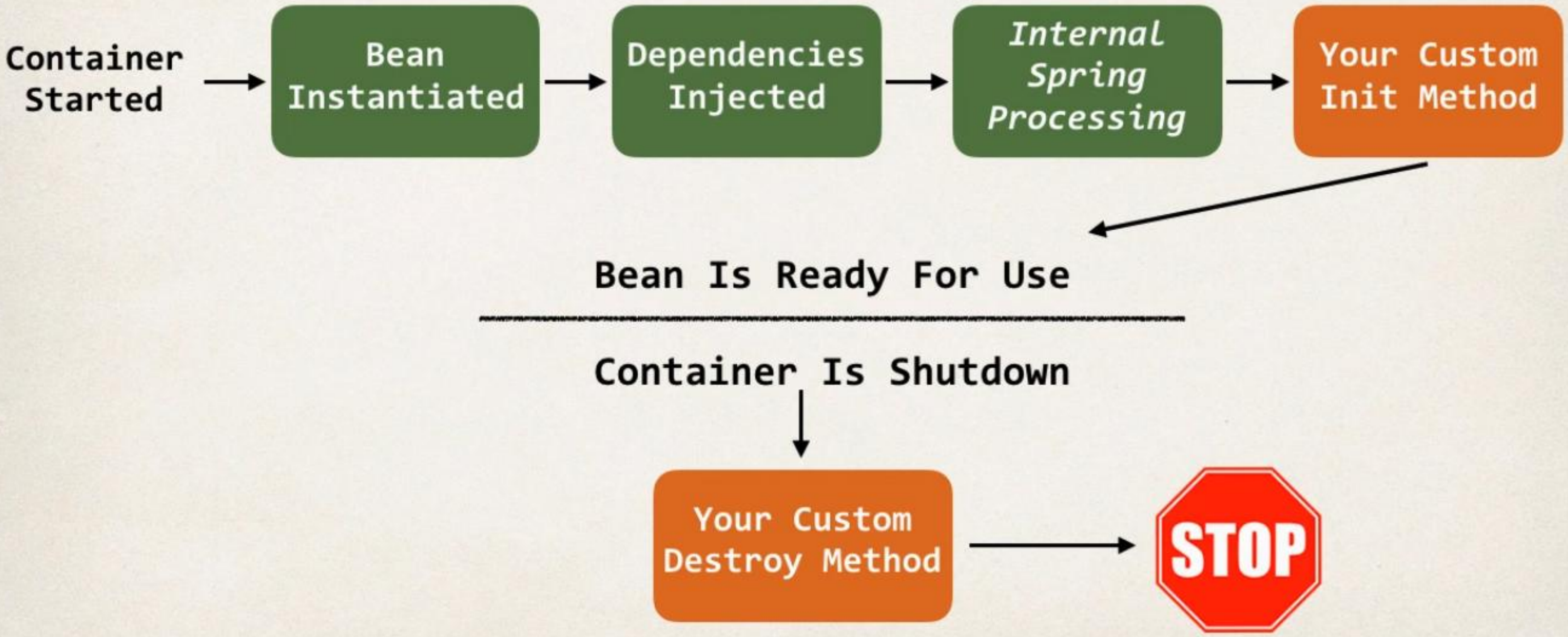


```xml
<bean id="vehiclebean" class="com.rit.p7di.Bike" scope="prototype">
</bean>
```

```
<terminated> MainDemo (6) [Java Application] C:\Users\Pradeep.Sudharsanan\.p
Are these same object : false
V1 object Ref : com.rit.p7di.Bike@2a798d51
V2 object Ref : com.rit.p7di.Bike@52bf72b5
```

## Bean Lifecycle Methods / Hooks

## Bean Lifecycle



## Bean Lifecycle Methods / Hooks

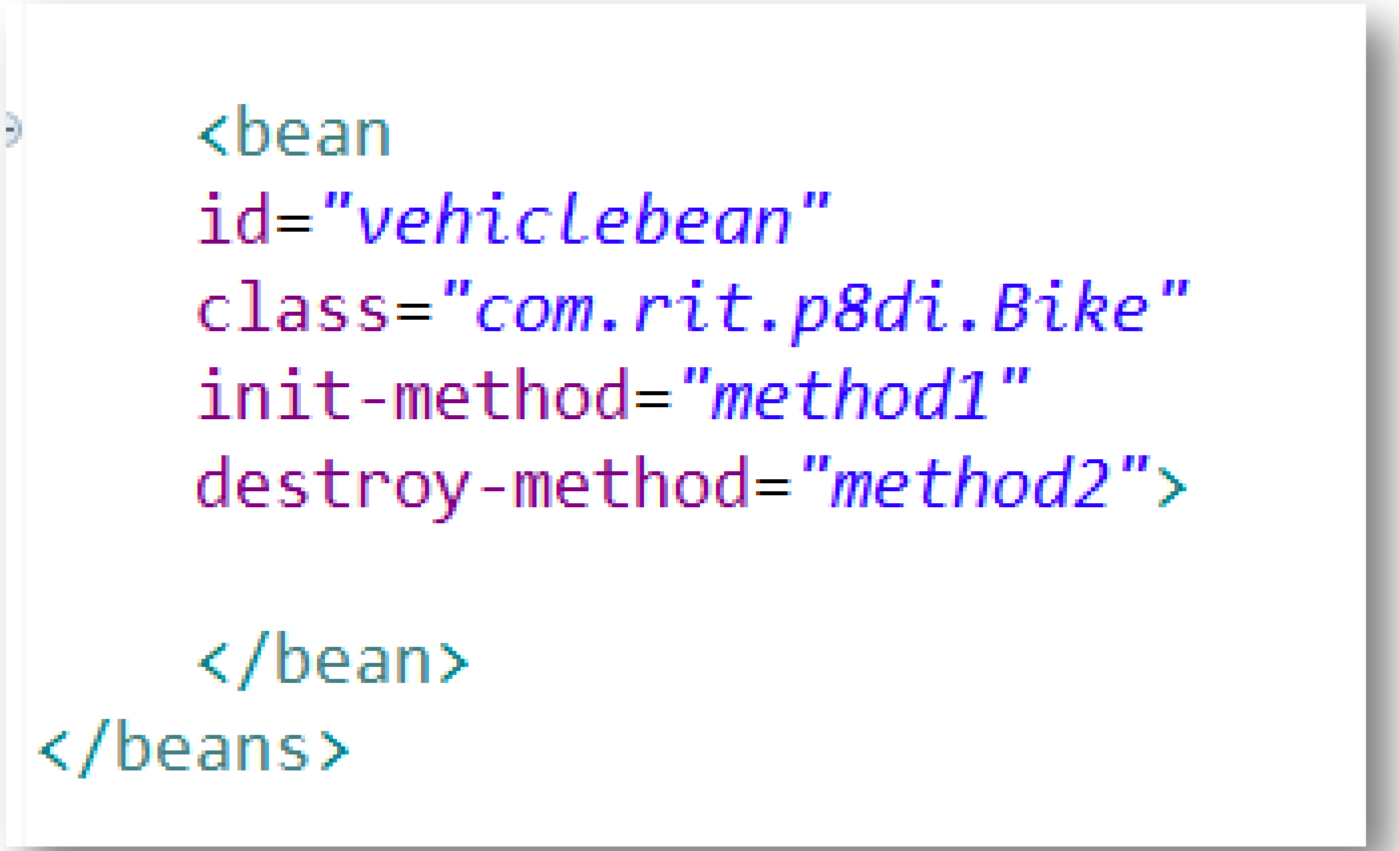
- You can add custom code during **bean initialization**
  - Calling custom business logic methods
  - Setting up handles to resources (db, sockets, file etc)

- You can add custom code during **bean destruction**
  - Calling custom business logic method
  - Clean up handles to resources (db, sockets, files etc)

## Development Process

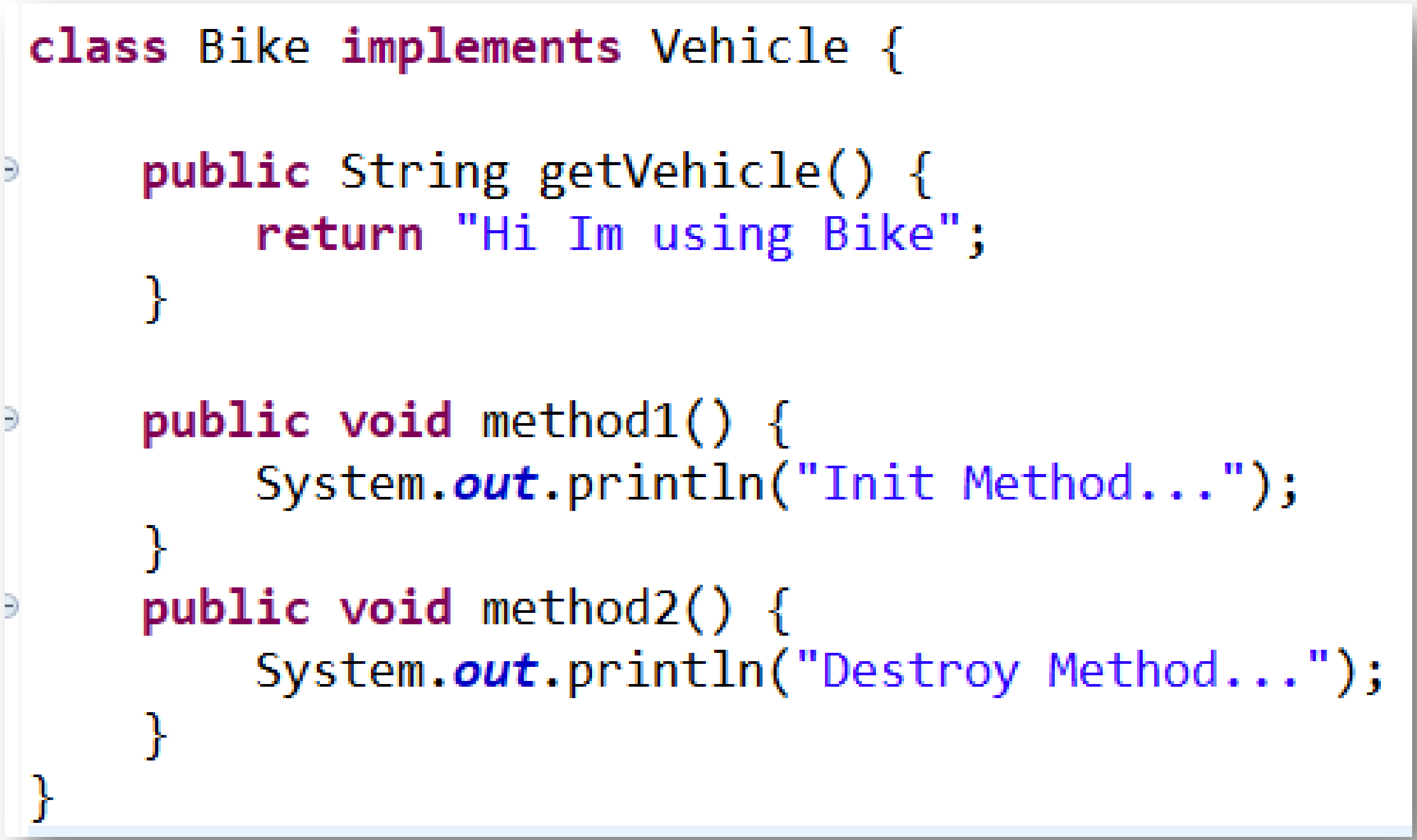
- Add init-method &destroy method in context file
- Add 2 lifecycle methods for init & destroy in Bike Class

## Context file



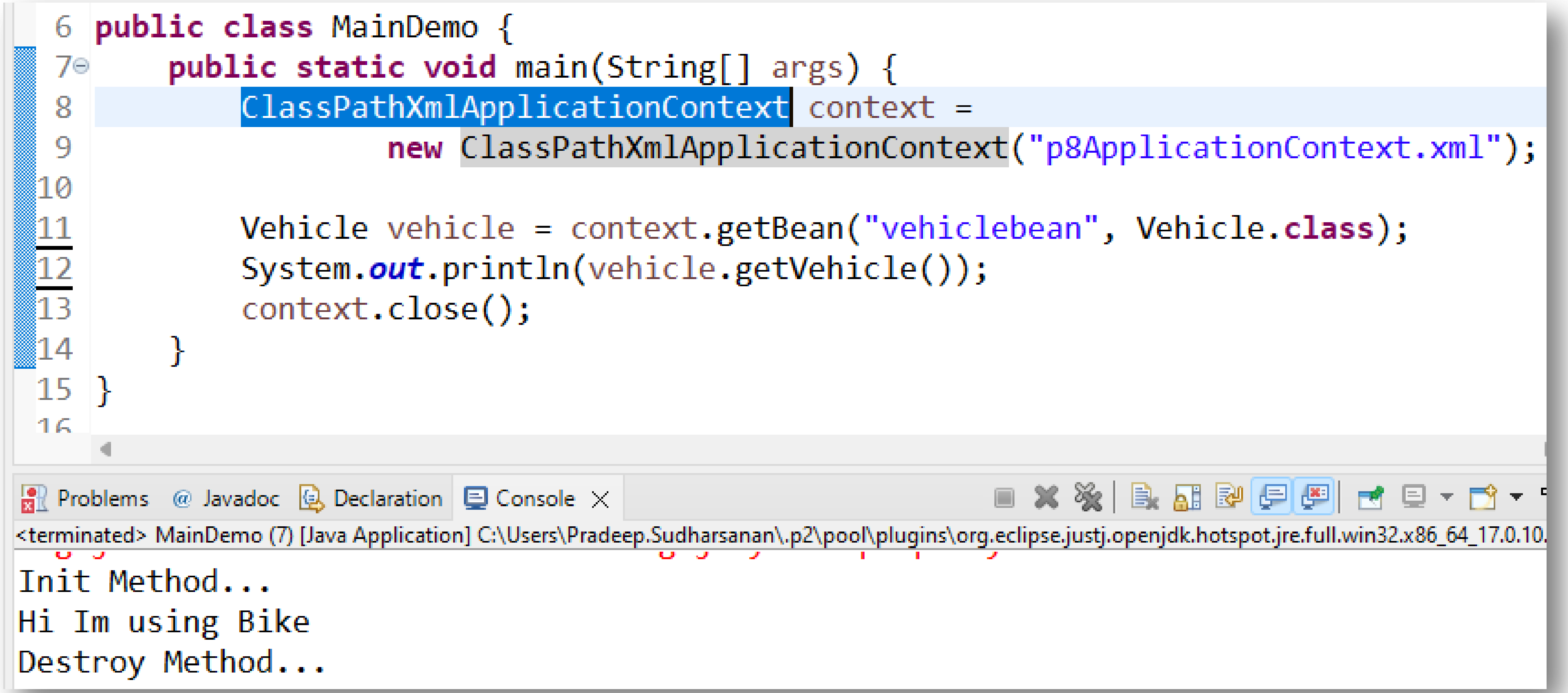
```
    <bean
    id="vehiclebean"
    class="com.rit.p8di.Bike"
    init-method="method1"
    destroy-method="method2">

    </bean>
</beans>
```

## Bike Class



```
class Bike implements Vehicle {

    public String getVehicle() {
        return "Hi Im using Bike";
    }

    public void method1() {
        System.out.println("Init Method...");
    }
    public void method2() {
        System.out.println("Destroy Method...");
    }
}
```

# Main Class



```
public class MainDemo {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context =
                new ClassPathXmlApplicationContext("p8ApplicationContext.xml");

        Vehicle vehicle = context.getBean("vehiclebean", Vehicle.class);
        System.out.println(vehicle.getVehicle());
        context.close();
    }
}
```

```
<terminated> MainDemo (7) [Java Application] C:\Users\Pradeep.Sudharsanan\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.
Init Method...
Hi Im using Bike
Destroy Method...
```

# Thank you