**Spring Learning**

Spring is solving the problem of dependency injection that world was facing. Dependency injection is nothing but, you want to introduce a dependency (object) on runtime.

Say, you want to build a Laptop, you can do as below.

**public** **class** Laptop {

**private** String name;

**private** String colour;

}

Now, let’s say that you need to add features to this like processor, and you don’t know about the processor at the time of building Laptop. The processor can be Intel or Amd. As below:

**public** **interface** Processor {

**public** **static** **final** String ***brand*** = "";

}

**public** **class** Intel **implements** Processor {

**public** **static** **final** String ***brand*** = "Intel";

}

**public** **class** AMD **implements** Processor {

**public** **static** **final** String ***brand*** = "AMD";

}

**public** **class** Laptop {

**private** String name;

**private** String colour;

Processor processor = **new** Intel();//This should be coming at runtime.

}

Your intention is not to create an object of Intel, but to get that injected. This is achieved by Spring Dependency Injection. You can also do this by following all the design patterns, but Spring offers this to you ready made.

Spring Boot gives all the basic configuration to the type of project that you are trying to build. Spring Boot follows **convention over configuration.**

Below is the class that got generated automatically.

@SpringBootApplication

**public** **class** MyfirstprojectApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(MyfirstprojectApplication.**class**, args);

}

}

Now, say you need to create an object of type Alien and it has a method.

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

**public** **void** doCode() {

System.***out***.println("I am coding...");

}

}

Now, say you need to call doCode() from MyFirstprojectApplicaion

@SpringBootApplication

**public** **class** MyfirstprojectApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(MyfirstprojectApplication.**class**, args);

Alien obj = **new** Alien();//If you do it, you are responsible to maintain it.

}

}

So, initialization should be left to Spring and the spring framework takes care of maintaining it, like creating, destroying etc.

**You can do this by below:**

Alien obj = getBean(Alien.**class**);

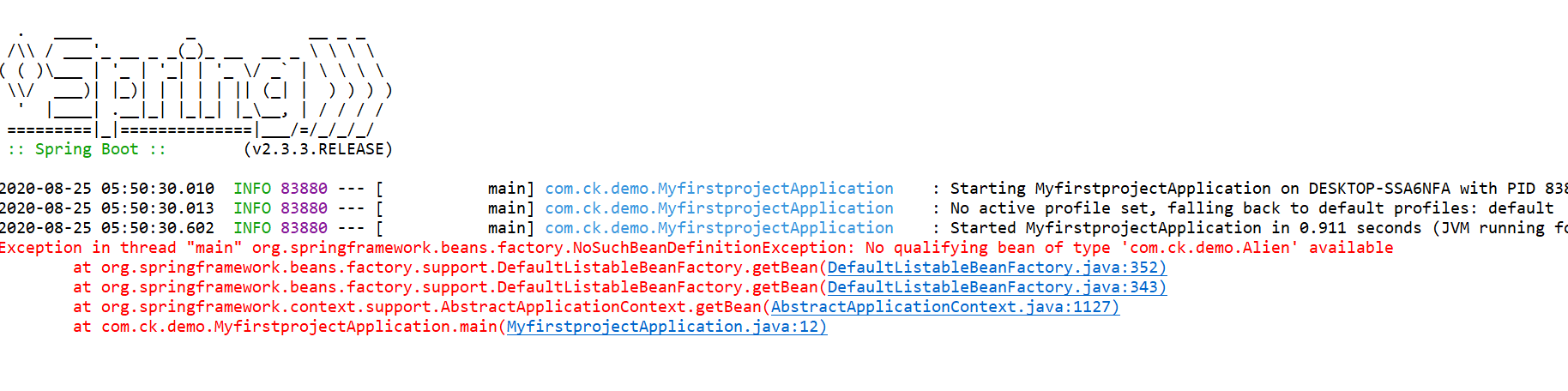
But, we need class of type ApplicationContext.java where getBean() is defined. We can get it from below:

ApplicationContext context = SpringApplication.*run*(MyfirstprojectApplication.**class**, args);

Alien obj = context.getBean(Alien.**class**);

obj.doCode();

If I run now by adding the above line instead of directly initializing, I get the below error.

So, I need to add the below to resolve the error.

@Component

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

**public** **void** doCode() {

System.***out***.println("I am coding...");

}

}

Now, below is the output:



Aliens need a laptop to code obviously. So, the below code can be added.

**public** **class** Laptop {

**public** **void** compile() {

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

}

}

@Component

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

Laptop lap;

**public** **void** doCode() {

lap.compile();

}

}

Now, we get the below error. Of course, we get as the object is not initialized.



This can be resolved using below:

@Component

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

@Autowired

Laptop lap;

**public** **void** doCode() {

lap.compile();

}

}

@Component

**public** **class** Laptop {

**public** **void** compile() {

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

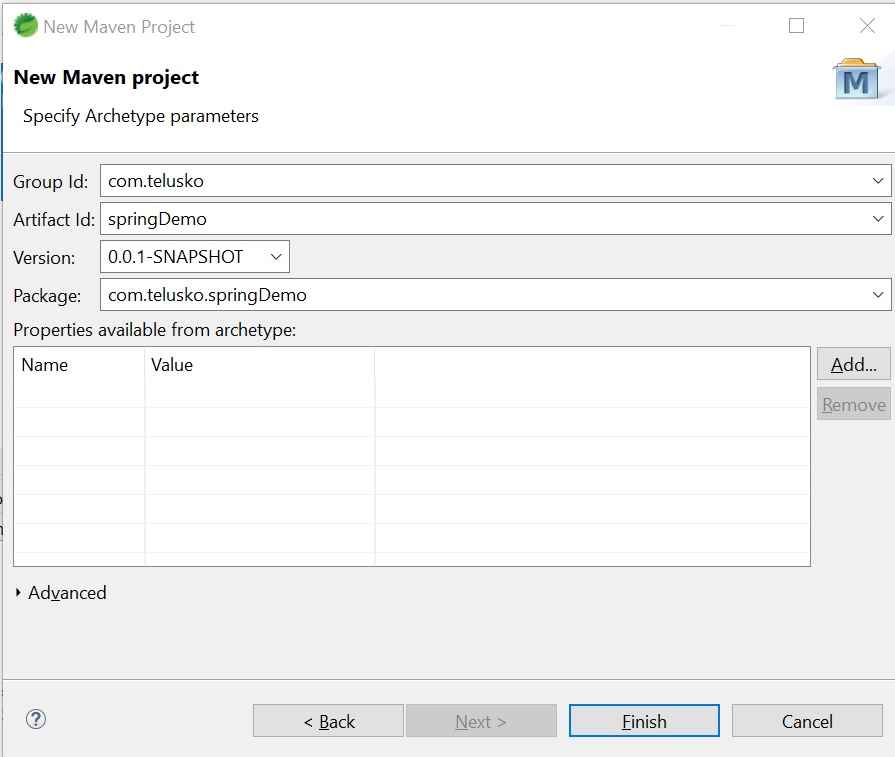
}

}

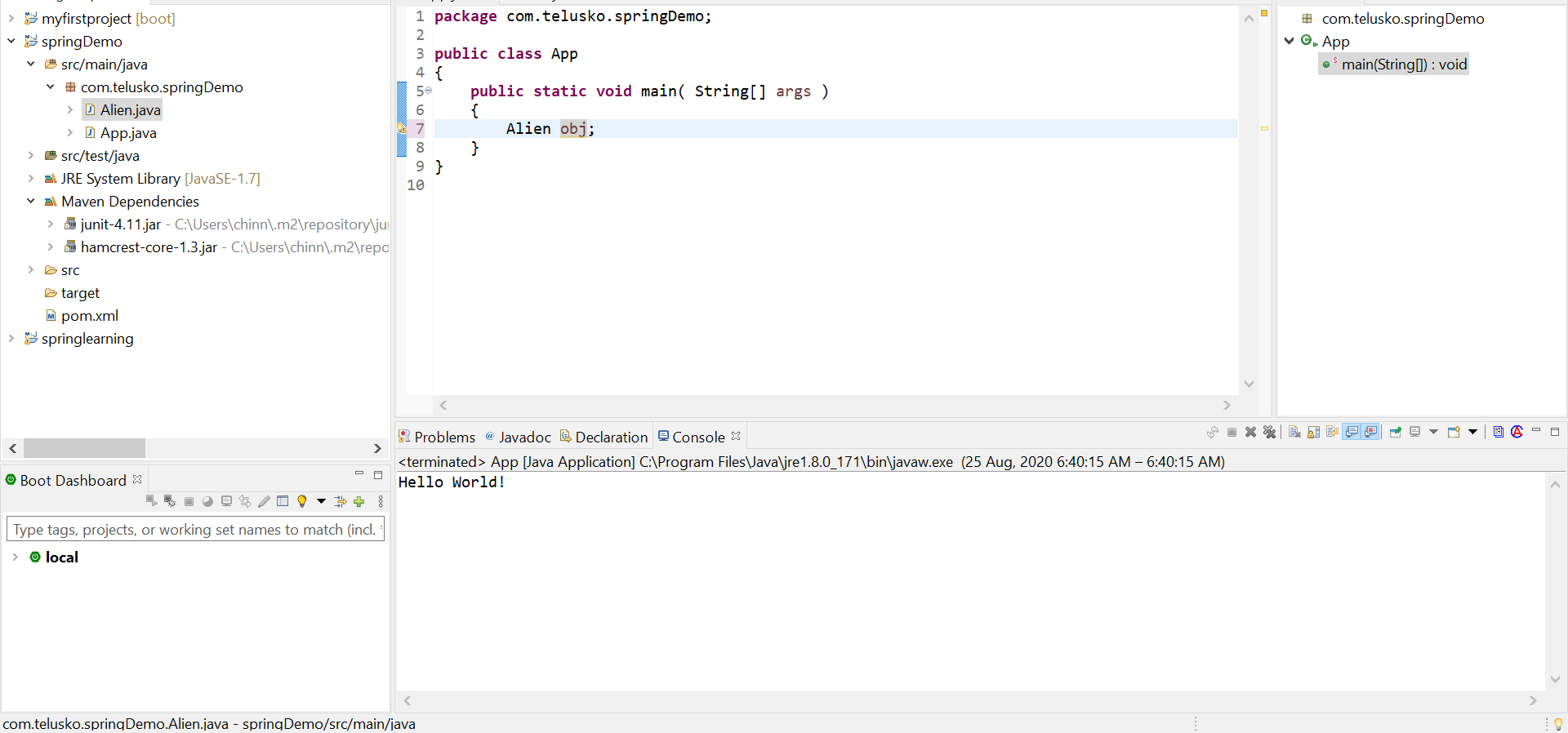


In the above, as we see, everything is done by Spring Boot with help of just the annotations. But since we are learning, we need to know what is happening internally. So, lets create a simple maven project and then see what happens internally.

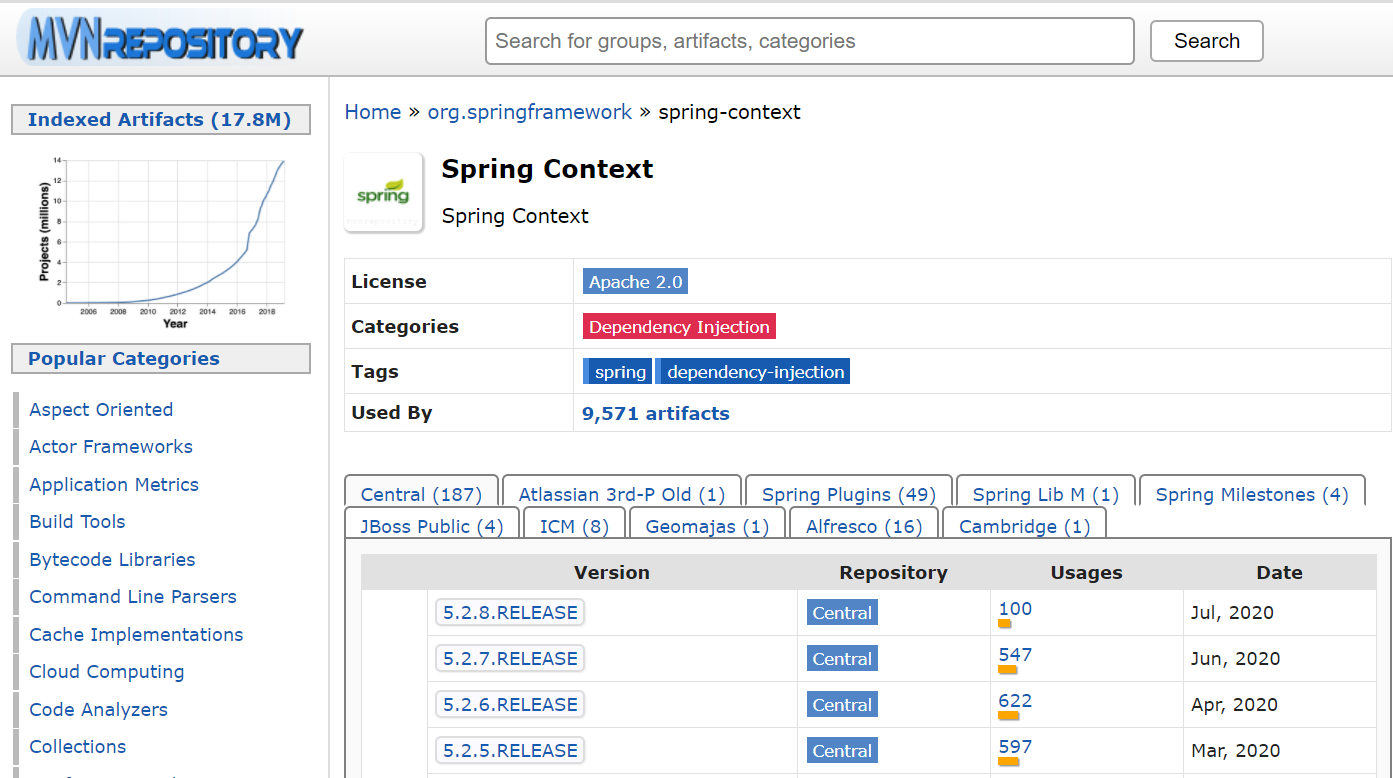
Click on Apache quick start project and name the project as below:



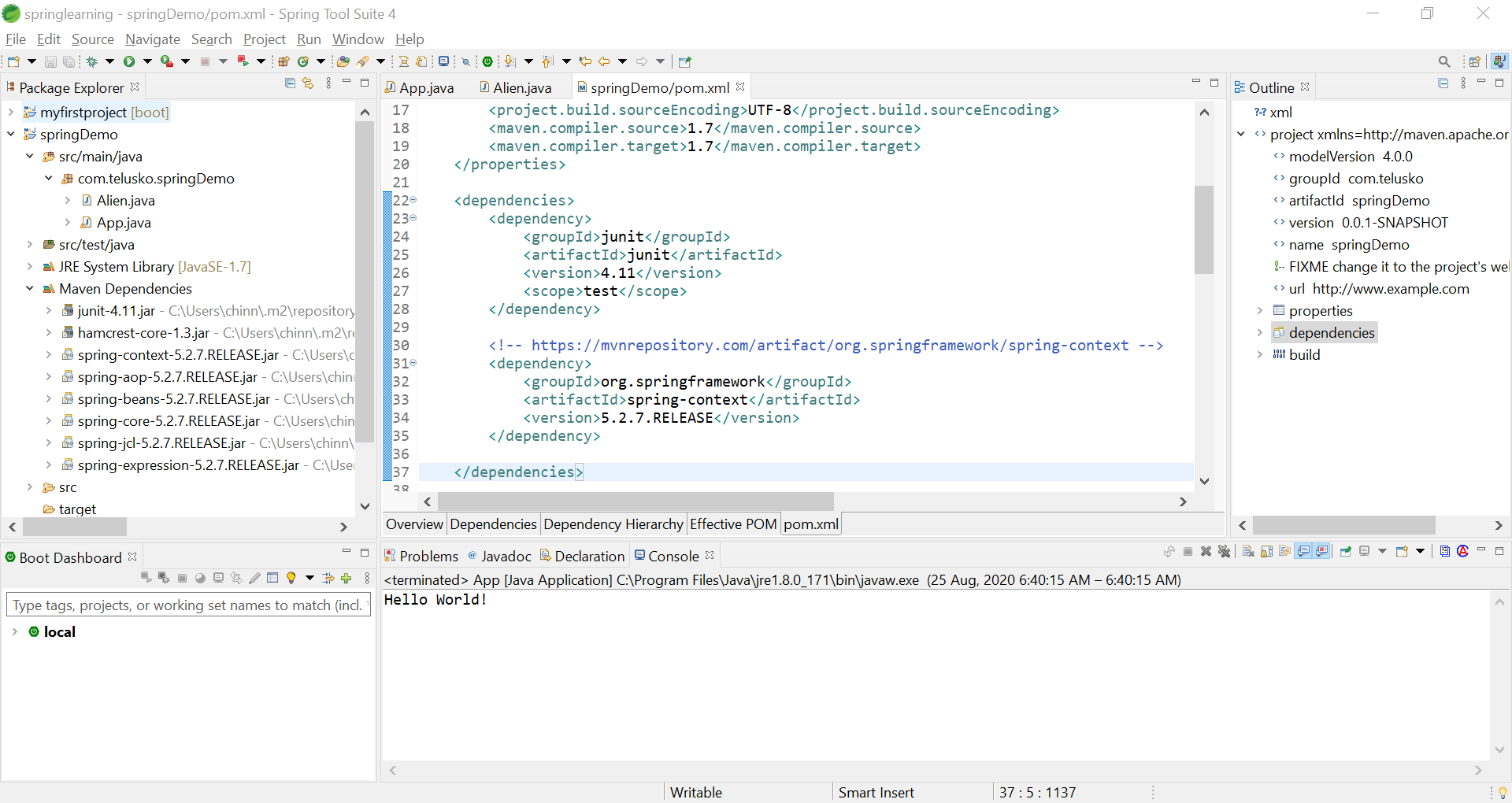
The below will be the output and folder structure created.



As you see above you do not find and Spring dependency. So, go to <https://mvnrepository.com/> and search as below



After adding the dependency, the below is seen in eclipse:



To make a car, we need car factory. To make a bike, we need a bike factory. Similarly, we need a bean factory to make bean.

You get factory object from below:

BeanFactory factory = **new** ~~XmlBeanFactory~~(**new** FileSystemResource("spring.xml"));

You call using below:

Alien obj = (Alien) factory.getBean("alien");

You see final code as below:

**public** **class** App

{

**public** **static** **void** main( String[] args )

{

BeanFactory factory = **new** ~~XmlBeanFactory~~(**new** FileSystemResource("spring.xml"));

Alien obj = (Alien) factory.getBean("alien");

}

}



By adding above config, we get the correct output. The line 8 says, hey spring, when you find some reference “alien”, you create an object of Alien.java class and return. We mentioned on line 9 with correct syntax.

If you see, the below is deprecated.

BeanFactory factory = **new** ~~XmlBeanFactory~~(**new** FileSystemResource("spring.xml"));

So, use the below which does the same thing.

ApplicationContext factory = **new** ClassPathXmlApplicationContext("spring.xml");

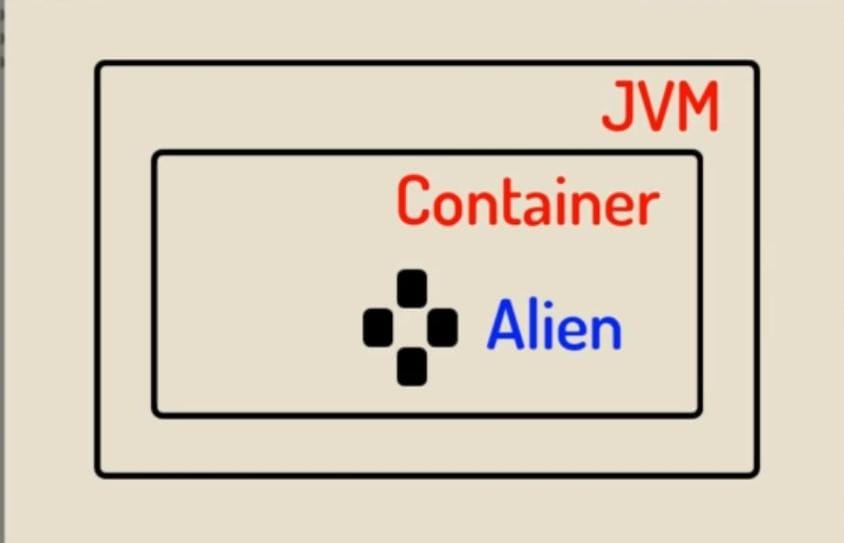
The only difference over the above is that we need to have spring.xml file in the class path, i.e. in the src folder. As the name suggests, ClassPathXmlApplicationContext("spring.xml") , it should be in the class path.

**What is a bean?**

Any class that has few variables and has getters and setters for those variables is called as a bean. As of now, assume, every class as a bean.

As learnt till now, Spring framework is responsible for creating object for whatever objects mentioned in spring.xml.

When below line is executed, all the objects mentioned in spring.xml are created in spring container. Spring container is present inside jvm as specified below.



So, whenever we want to create a new object called Alien, spring searches for it in spring container and returns the same. Like shown above in the image.

**When we create a constructor in Alien class, and have something printed, we will always see that the statement is printed. With this, we can conclude that the object is always created irrespective of we are using it or not.**

**By default, objects created in Spring are singleton. This can be proved from below.**

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

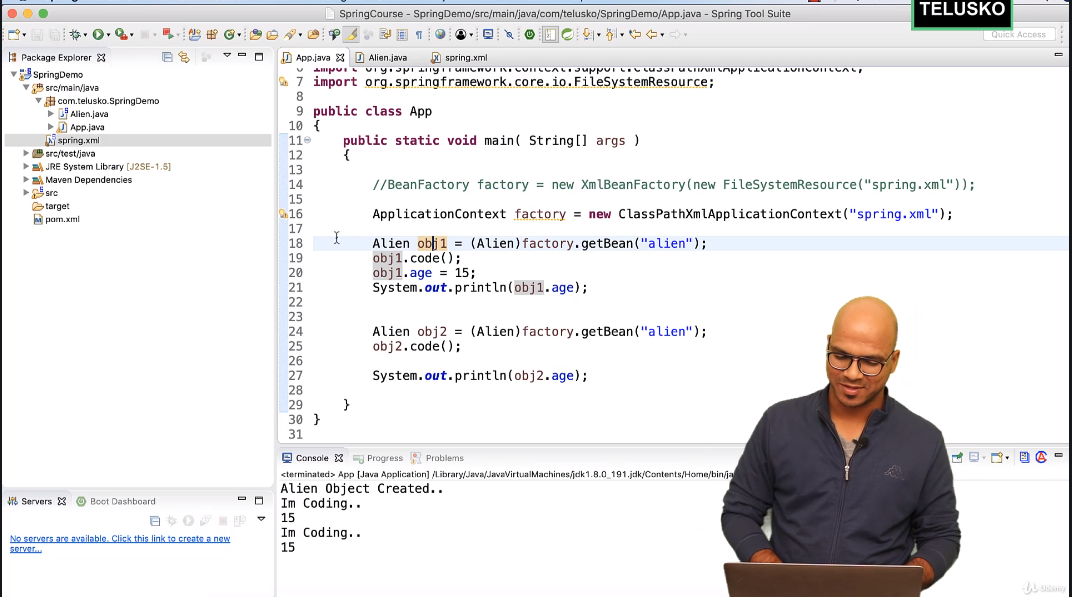
**priva**te **int** age;

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

System.***out***.println("I am coding...");

}

}



As you see, both objects are printing the same value. So, with this we can conclude the objects are the same and not different.

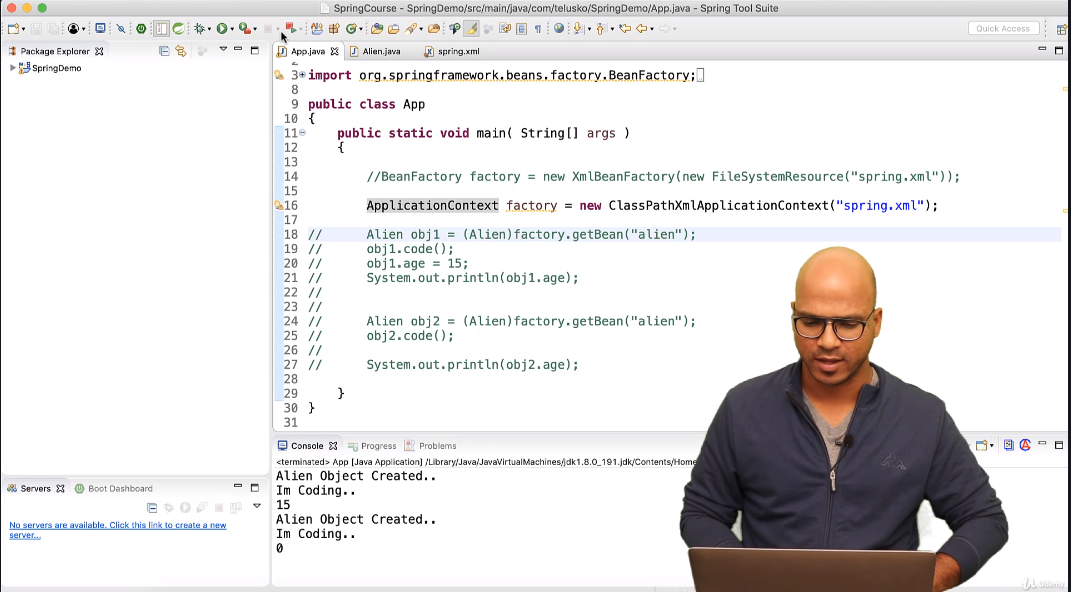
**What if you do not want to have a singleton object?**

By default, the scope of a spring object is **singleton**. The below is by default present in spring.

<bean id=*"alien"* class=*"com.telusko.springDemo"* scope=*"singleton"*></bean>

When the scope is **prototype**, spring creates a new object every time you ask for an object.

<bean id=*"alien"* class=*"com.telusko.springDemo"* scope=*"prototype"*></bean>



The catch over here is that, if the scope is **prototype**, spring will not create any object unless the object is asked for, unlike in case of **singleton** where **constructor** is called even though the object is never created by programmer. The above screenshot has the code commented, and if this code is run, there will be no output on console 😊

**Setter injection: -**

In POJO, we have all properties in the class to be private and every property has a public setter and a public getter as mentioned below.

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

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

**public** Alien(**int** age) {

**super**();

**this**.age = age;

}

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

**return** age;

}

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

**this**.age = age;

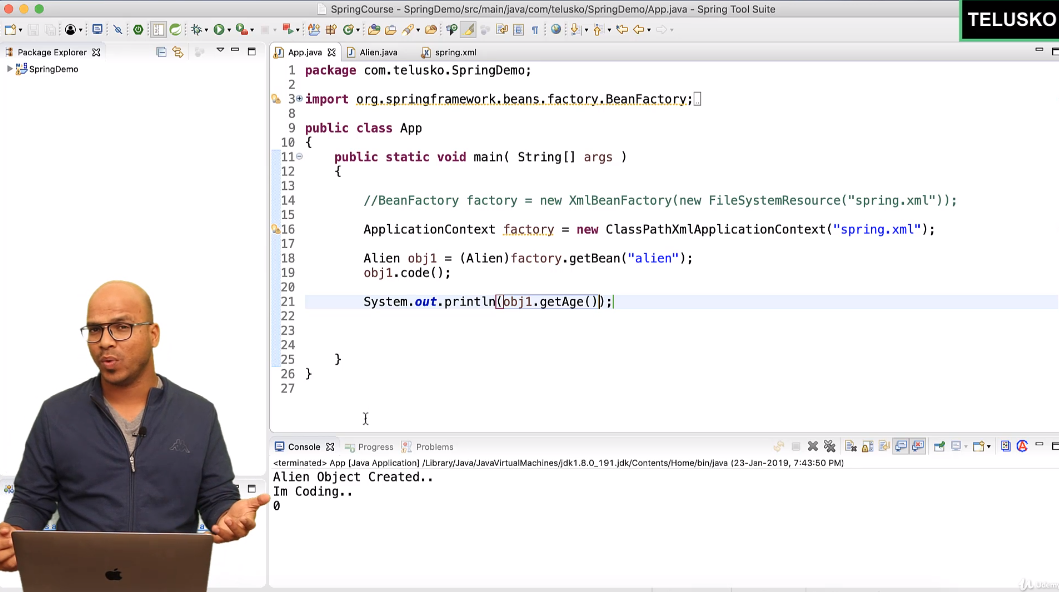
}

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

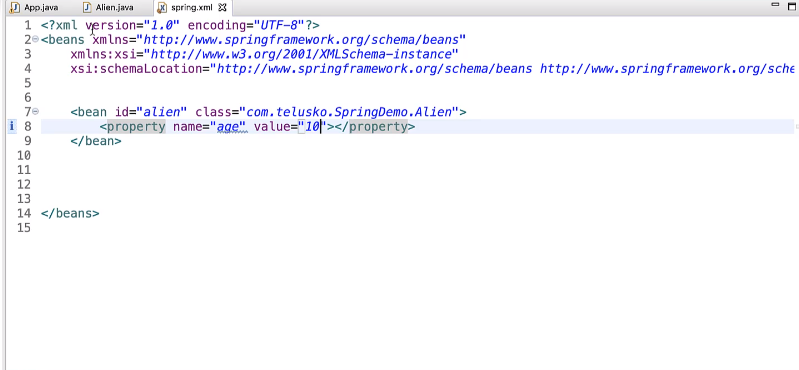
System.***out***.println("I am coding...");

}

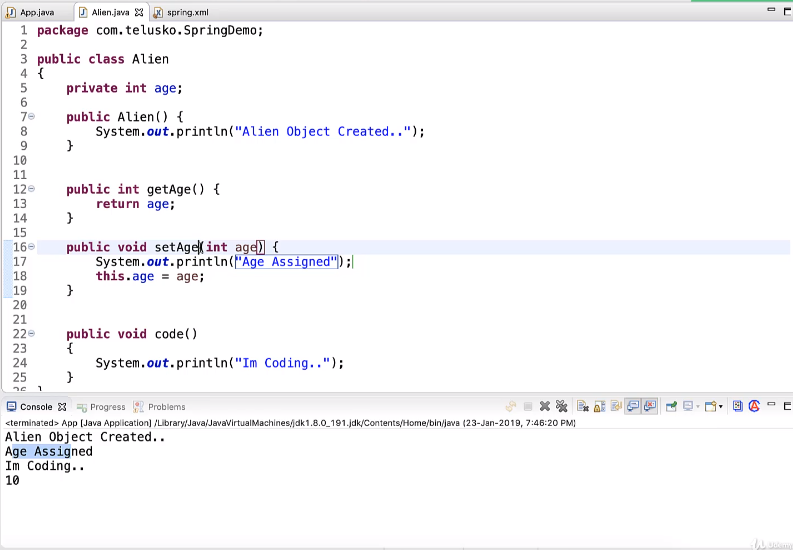
}



As see above, by default, the value is 0. What if you want to specify the default value to be other, you can do that in the Alien class. The other way is by providing it in the spring.xml file using below.



When you mention the words, property and value, spring by default calls the setter method for the property mentioned in configuration. This can be proved by below:



By using property and value, we are able to assign values to primitive variables. What if the Reference is used instead of primitive?

**public** **class** Laptop {

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

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

}

}

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

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

**private** Laptop laptop;

**public** Alien(**int** age) {

**super**();

**this**.age = age;

}

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

**return** age;

}

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

System.***out***.println("Age assigned...");

**this**.age = age;

}

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

System.***out***.println("I am coding...");

}

**public** Laptop getLaptop() {

**return** laptop;

}

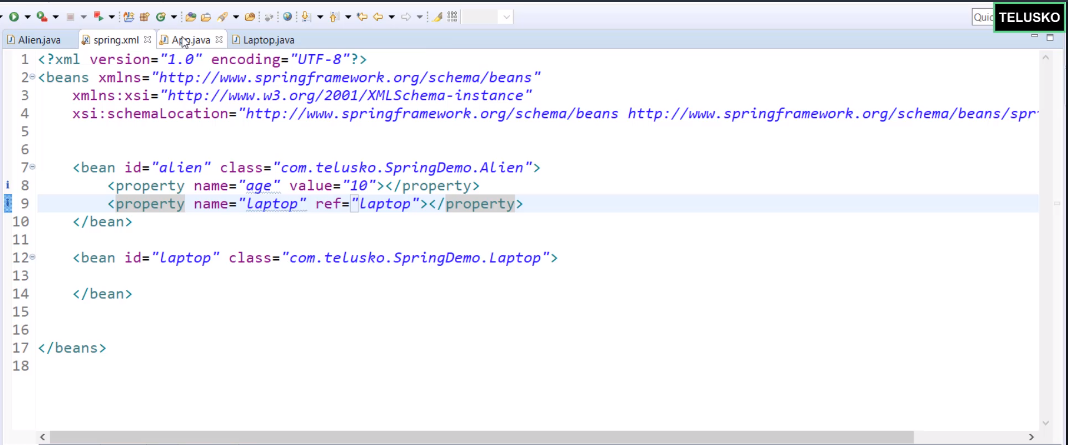
**public** **void** setLaptop(Laptop laptop) {

**this**.laptop = laptop;

}

}

The below is the change in spring.xml file since you are using a reference type and not a primitive type. So, **value** gets replaced by **ref**.



We can initialize the properties of a bean by using constructor as well. Say, you have a constructor using only one property, say **age**. The below is the change in the **spring.xml**.

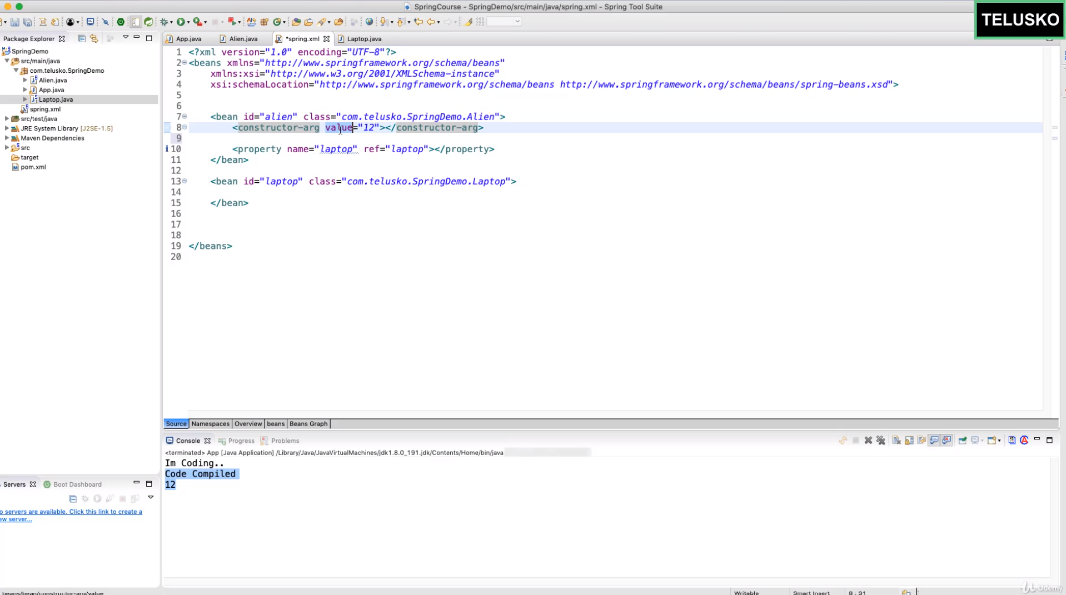
Instead of **ref**, we use **constructor-arg**

**Difference between constructor and setter injection: -**

**Partial dependency:** can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.

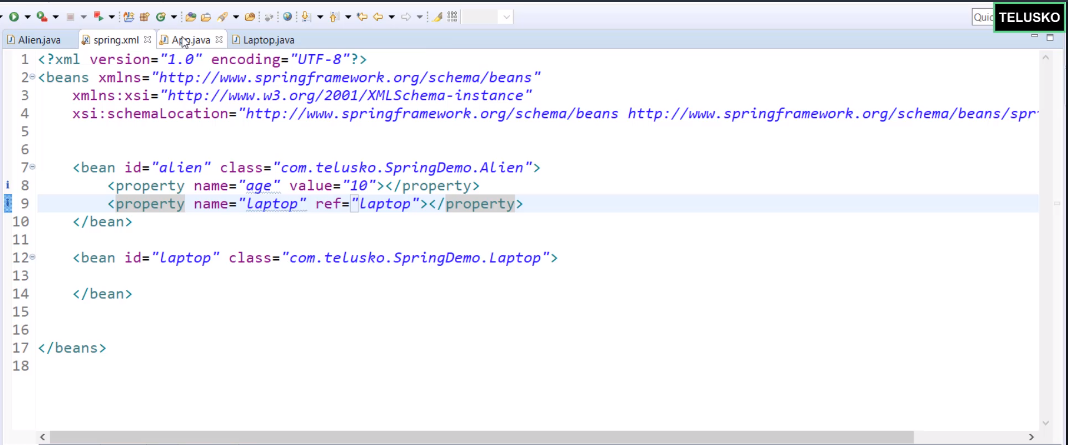
**Overriding:** Setter injection overrides the constructor injection. If we use both constructor and setter injection, **IOC container will use the setter injection**.

**Changes:** We can easily change the value by setter injection. It doesn’t create a new bean instance always like constructor. So setter injection is flexible than constructor injection.



**Autowiring: -**

As per Alien class, Every Alien needs a Laptop to code. We specify the same in the spring.xml file as below.



But, Alien can code using a laptop or a desktop right. So, say the below is the code structure now.

**public** **interface** Computer {

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

}

**public** **class** Laptop **implements** Computer{

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

System.***out***.println("Code compiled in a Laptop...");

}

}

**public** **class** Desktop **implements** Computer{

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

System.***out***.println("Code compiled in a Desktop...");

}

}

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

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

**private** Computer com;

**public** Alien(**int** age) {

**super**();

**this**.age = age;

}

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

**return** age;

}

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

System.***out***.println("Age assigned...");

**this**.age = age;

}

**public** Computer getCom() {

**return** com;

}

**public** **void** setCom(Computer com) {

**this**.com = com;

}

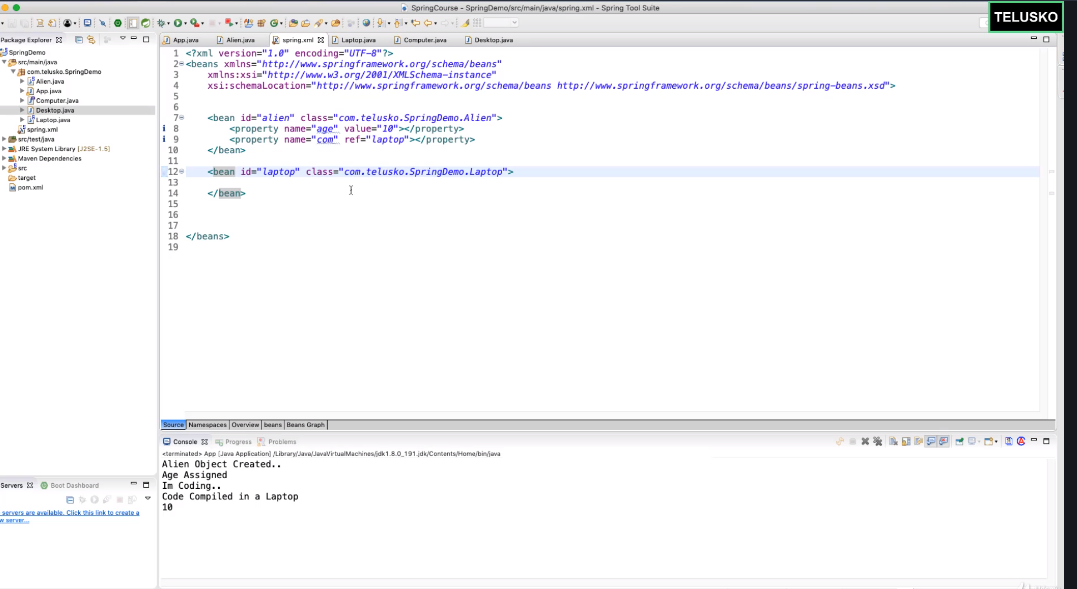
**public** **void** code() {

com.code();

System.***out***.println("I am coding...");

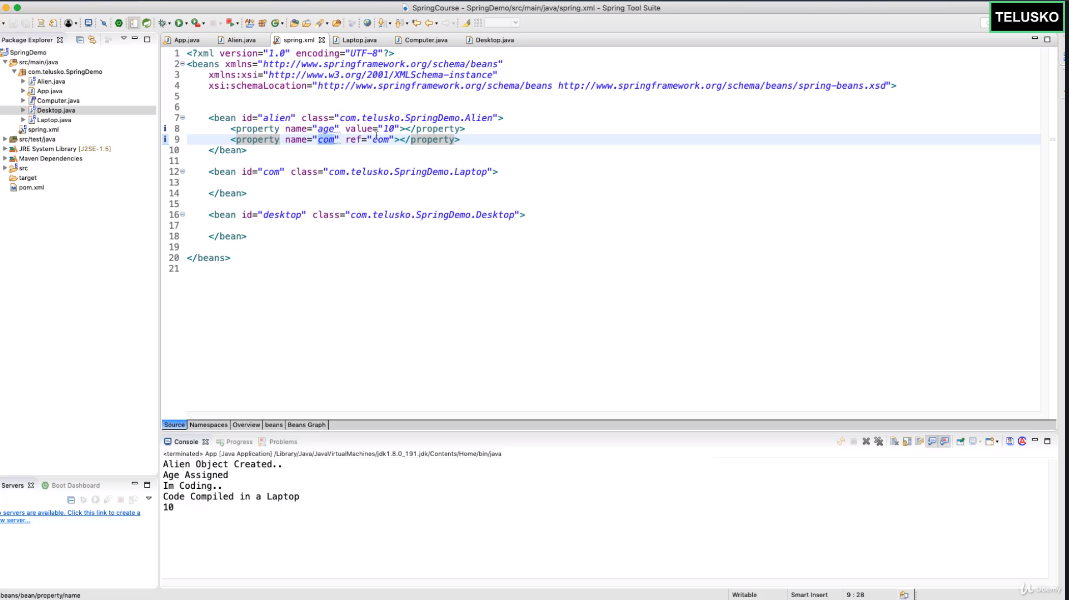
}

}

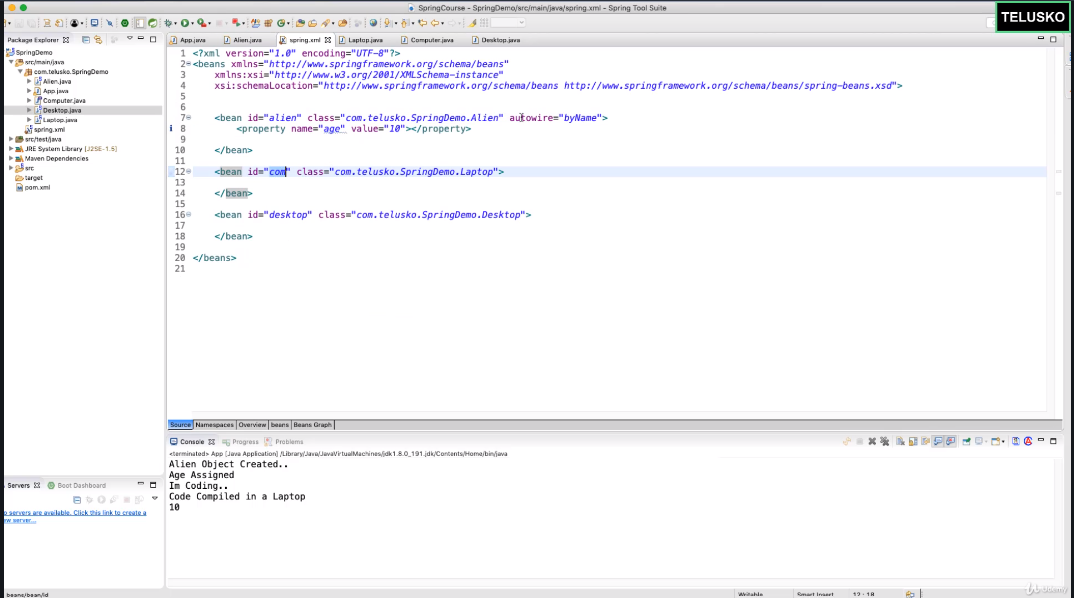


The above works because of the **name** specified in the **spring.xml**

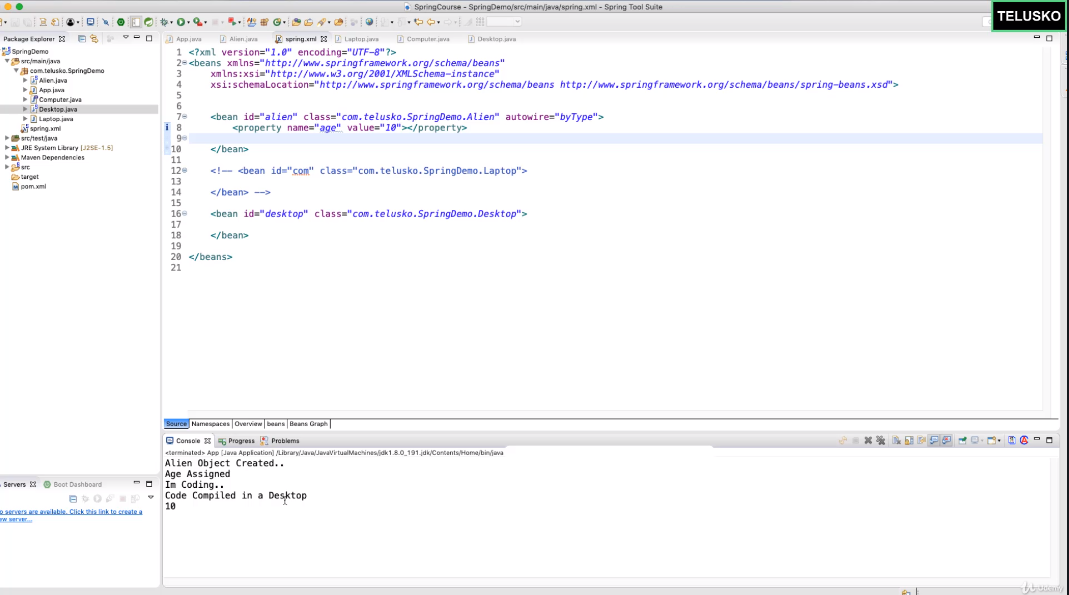
**Say, I make the below change, but the code still works.**



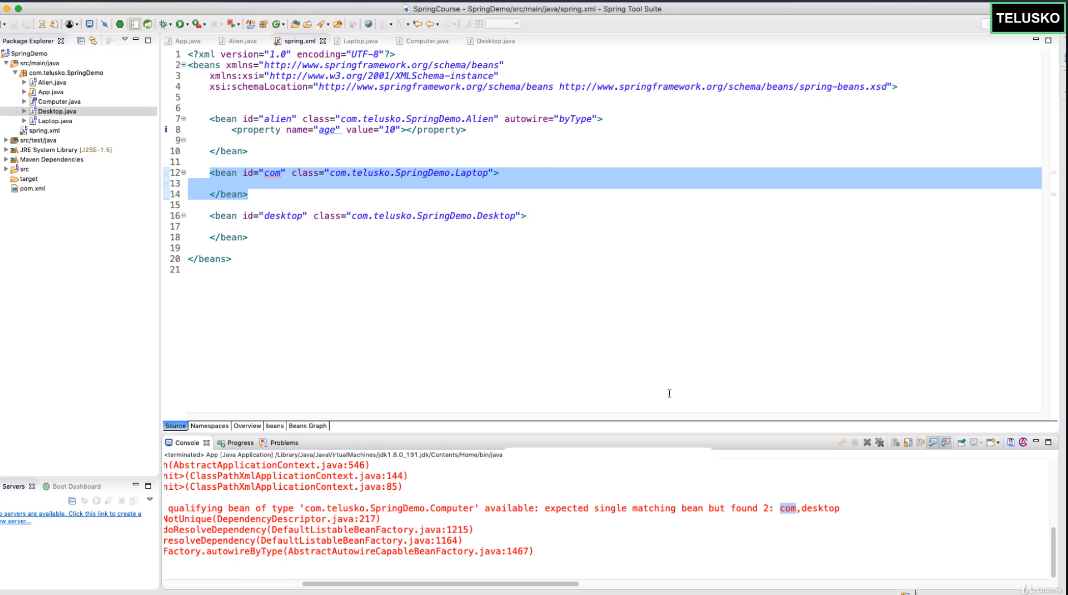
So, straight question is, if name and ref is the same and it still works, so, why to specify explicitly on line 9. You can remove this right. But still I need to connect the variable present in the java class to the bean definition. How can I do that? I can achieve this using auto wiring. **Just add the below to spring.xml.**



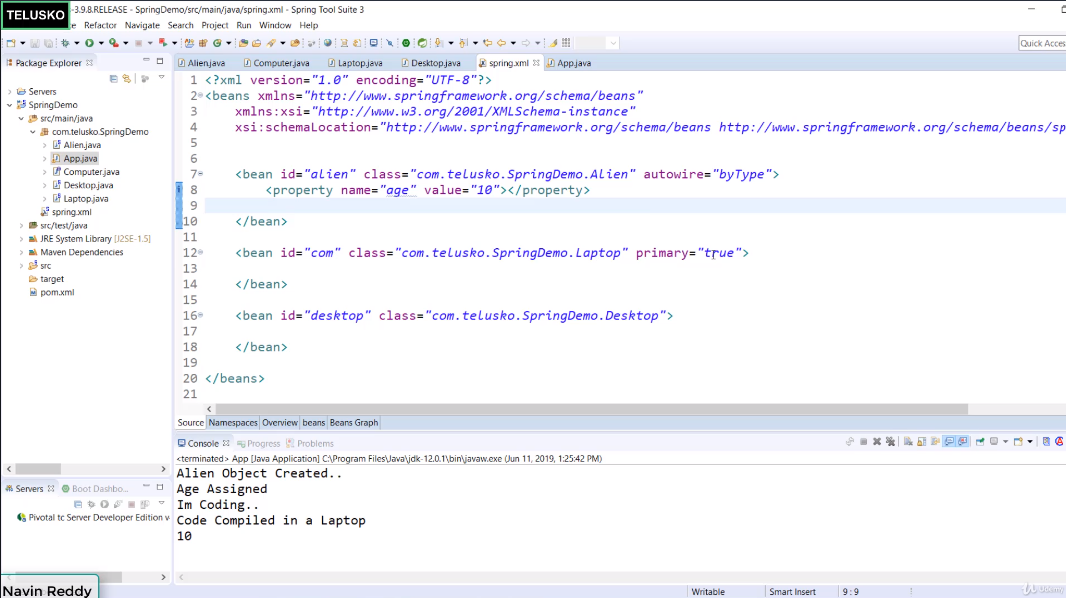
The above is the beauty of Auto Wire. You can also do this using **byType**



Say, you have both types and now, spring confuses as on what basis it needs to choose, should it choose the first one specified or what? The below is the confusion.



**How to solve this confusion? Use “*primary*”**

****