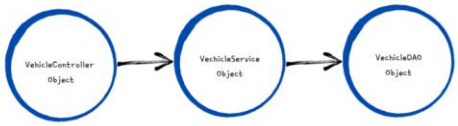
INTRODUCTION TO BEANS WIRING INSIDE SPRING

eazy bytes

- *Inside Java web applications, usually the objects delegate certain responsibilities to other objects. So in this scenarios, objects will have dependency on others.*
- *In very similar lines when we create various beans using Spring, it our responsibility to understand the dependencies that beans have and wire them. This concept inside is called Wiring/Autowiring.*



If you take any web application inside Java, usually the objects will delegate certain functionality or responsibilities to other objects inside web applications.
So we will never write all the business logic inside a same class because it will be super, super complex to maintain in the future.
That's why from day one of Java we will try to segregate all our business logic based upon our need into the corresponding layers.
Like inside our web applications.We usually maintain a service layer, controller layer database persistent layer and frontend layer.

Like if you take the example that I'm showing on the screen, there is a transaction happening at the UI of the web application. Maybe the user is trying to enter his vehicle details and he's trying to save his vehicle details into the database.

So in this scenarios, first we have a controller layer which will accept the requests from the UI and process all the validations.
Once all the validations are passed, it will hand over that input received from the user to the service layer and inside vehicle service object.

We can write our core business logic. Maybe you want to validate is there any other vehicle already registered with the same details?

So those kind of business validations or any other business logic we write inside the vehicle service class or in the service layer of your web application.

And once all your business logic is executed, definitely you want to store the data of an vehicle or any input that is provided by the user into a database.So to perform CRUD operations onto the database, we have a vehicle DAO object or a DAO layer inside our web applications which will take care of persisting the data, updating the data, deleting the data based upon our requirements.

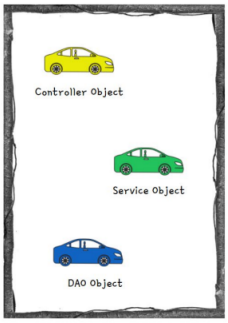
So in this flow, vehicle controller object has a dependency or it is handing over certain responsibilities to the service object and the same applicable for service object also it is handover certain responsibilities to the data object.

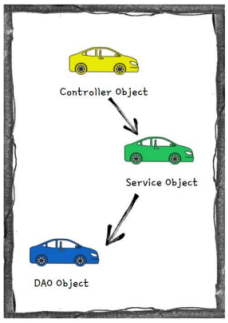
INTRODUCTION TO BEANS WIRING INSIDE SPRING

SPRING CONTEXT WITH OUT WIRING

SPRING CONTEXT WITH WIRING & DI

Beans are present but there is no wiring present b/w them





Now I have done the wiring configurations inside my web application.

That way my spring IOC container now is smart enough to understand the dependencies that all these objects has in between them, and with the help of wiring configurations, it will do dependency injection when my code executes.
Like whenever I'm trying to execute the code in the controller object.
By the time my code is looking for the service object, my spring framework will make sure it is injecting the service object.
That way my code will not have null pointer exception.Similarly, inside service object class.
My spring framework will make sure a Java object bean is injected based upon the wiring configurations

NO WIRING SCENARIO INSIDE SPRING

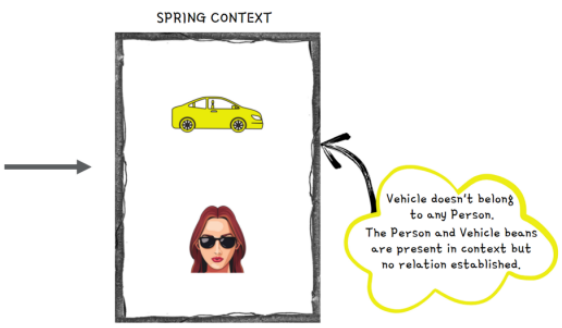
eazy bytes

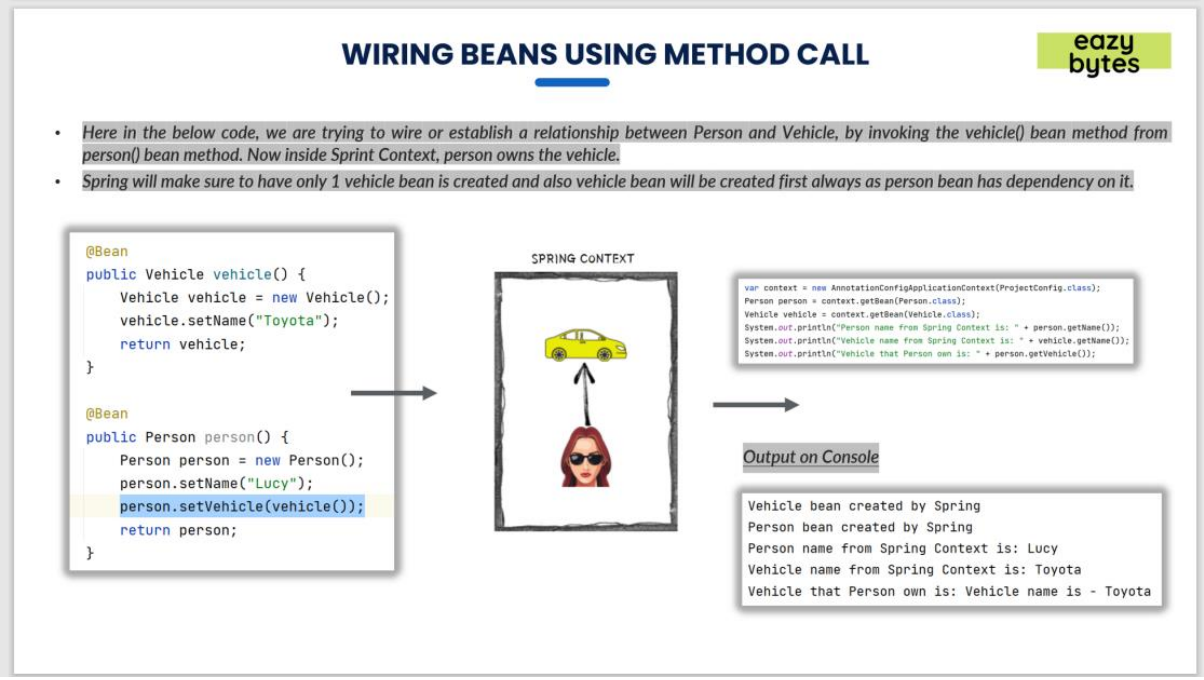
Consider a scenario where we have two java classes Person and Vehicle. The Person class has a dependency on the Vehicle. Based on the below code, we are only creating the beans inside the Spring Context and no wiring will be done. Due to this both this beans present inside the Spring context with out knowing about each other.

```java
public class Vehicle {

    private String name;
```

```java
public class Person {

    private String name;
    private Vehicle vehicle;
```

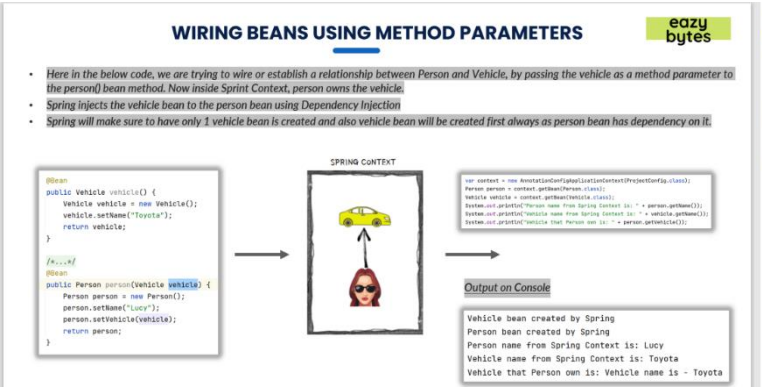
```java
@Bean
public Vehicle vehicle() {
    Vehicle vehicle = new Vehicle();
    vehicle.setName("Toyota");
    return vehicle;
}

@Bean
public Person person() {
    Person person = new Person();
    person.setName("Lucy");
    return person;
}
```





Example 10:

31.Wiring Beans using Method Parameters



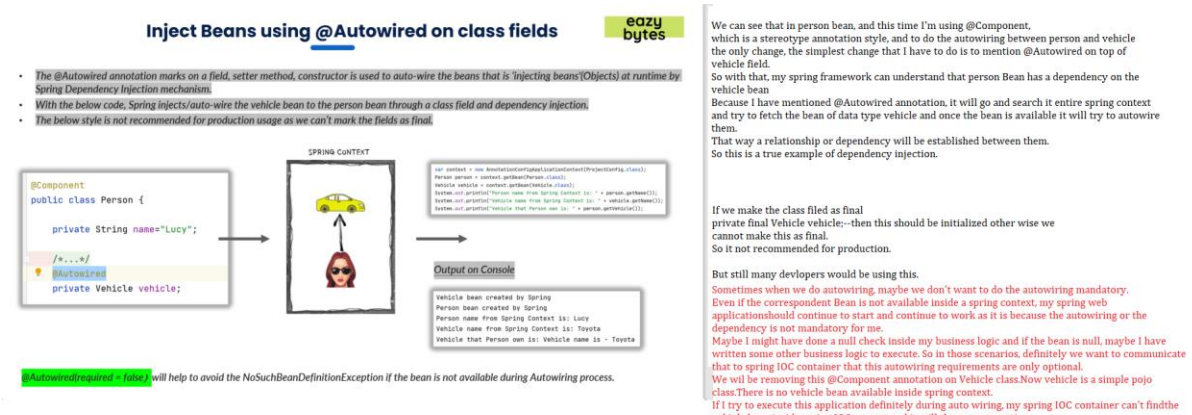
Now the person method, which is annotated with @Bean is accepting a method parameter and the method parameter is of type vehicle.

So with this now my spring framework is smart enough to understand.So this person whom I'm going to create a bean has also have an dependency on vehicle bean because it can clearly look at the method parameters.

So spring framework, whenever it sees a method parameter, it will try to look if there is a bean with the same data type inside the spring context.

Definitely there is a spring bean which is of type vehicle data type, so it will try to take that and inject or autowire automatically.

32.Wiring Beans using @Autowired on class fields.



We can see that in person bean, and this time I'm using @Component,
which is a stereotype annotation style, and to do the autowiring between person and vehicle the only change, the simplest change that I have to do is to mention @Autowired on top of vehicle field.
So with that, my spring framework can understand that person Bean has a dependency on the vehicle bean
Because I have mentioned @Autowired annotation, it will go and search it entire spring context and try to fetch the bean of data type vehicle and once the bean is available it will try to autowire them.
That way a relationship or dependency will be established between them.
So this is a true example of dependency injection.

If we make the class filed as final
private final Vehicle vehicle;--then this should be initialized other wise we cannot make this as final.
So it not recommended for production.

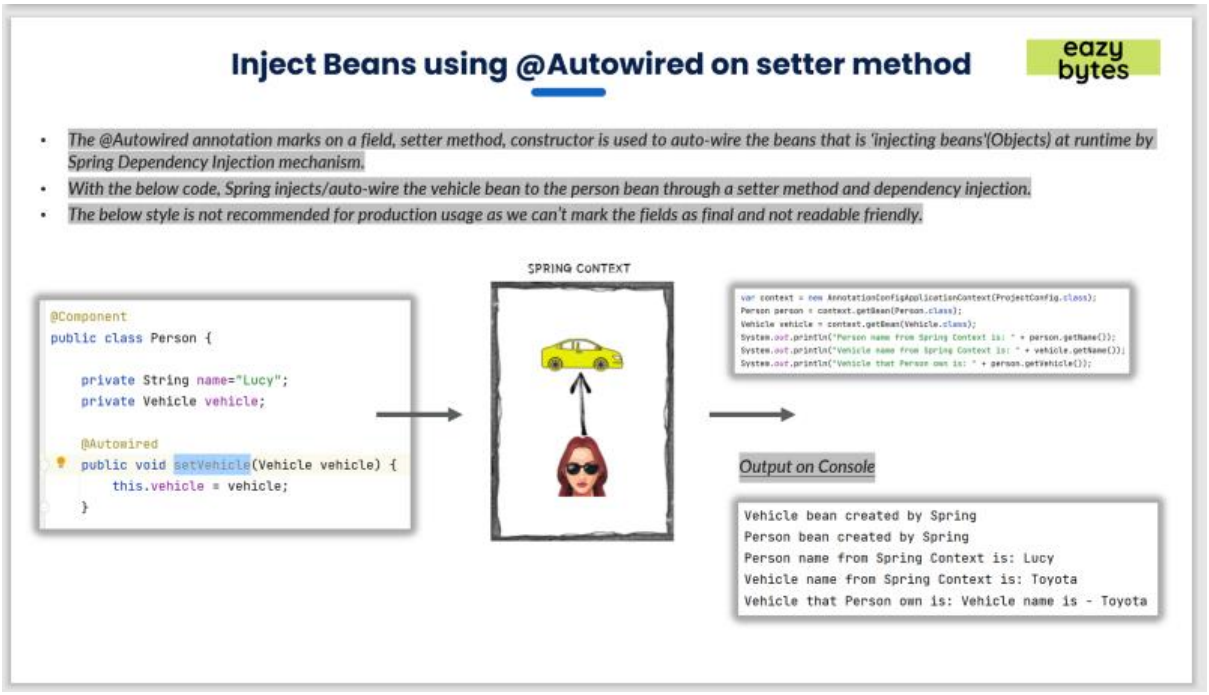
But still many devlopers would be using this.
Sometimes when we do autowiring, maybe we don't want to do the autowiring mandatory.
Even if the correspondent Bean is not available inside a spring context, my spring web applicationshould continue to start and continue to work as it is because the autowiring or the dependency is not mandatory for me.
Maybe I might have done a null check inside my business logic and if the bean is null, maybe I have written some other business logic to execute. So in those scenarios, definitely we want to communicate that to spring IOC container that this autowiring requirements are only optional.
We wil be removing this @Component annotation on Vehicle class.Now vehicle is a simple pojo class.There is no vehicle bean available inside spring context.
If I try to execute this application definitely during auto wiring, my spring IOC container can't findthe vehicle bean inside spring IOC context and it will throw an exception.

So you can see we received an exception saying that no such bean definition exception of type vehicle expected at least one bean which qualifies as autowire candidate, because we use autowired annotation and by default the value required is true.

So to handle these kind of scenarios, what we can do is post autowired annotation we can mention required is equal to false.

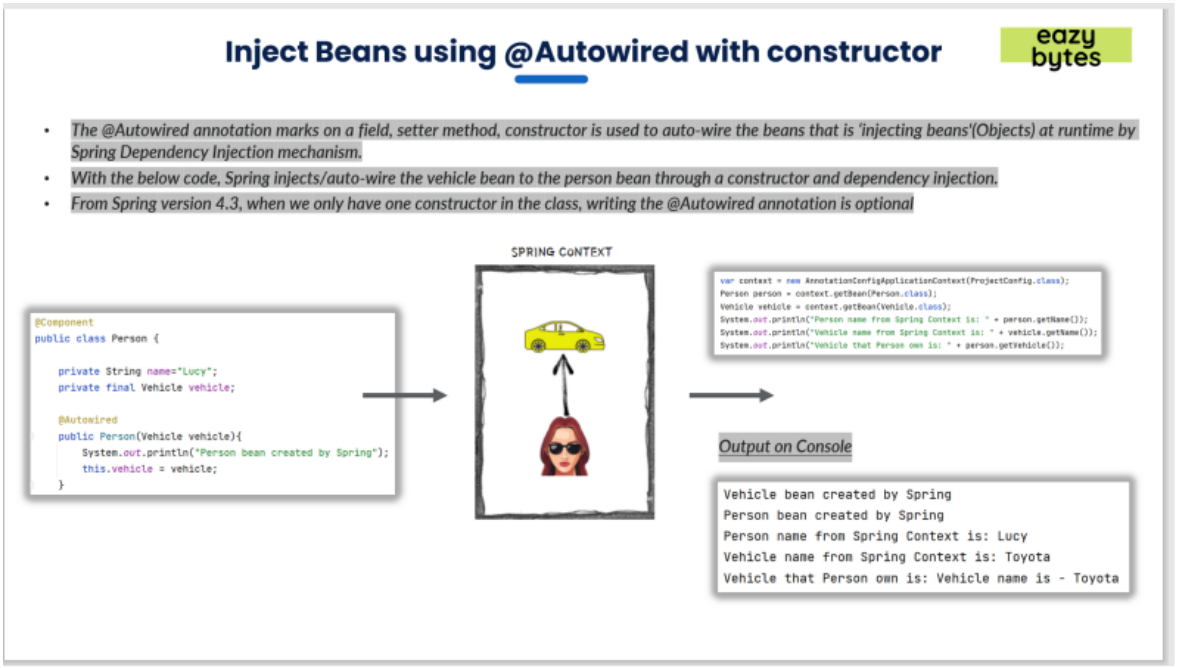
Example 12

33.Injecting Beans using @Autowired on setter method.



Example 13

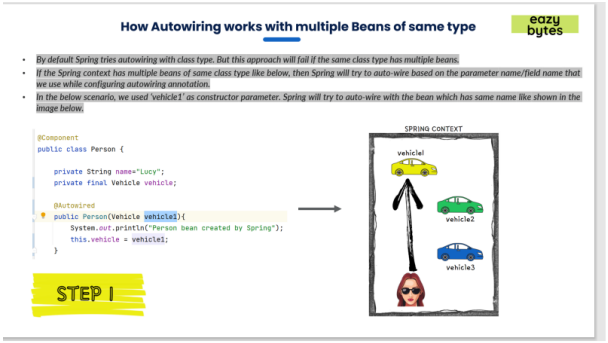
34:Injecting Beans using @Autowired on Constructor:

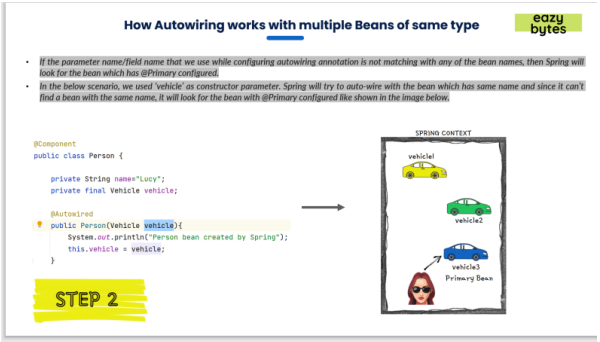


So this way once the vehicle bean is created and injected into person bean, then there is no way of changing the values inside it, especially if you are super serious about security concerns inside your code and you don't want any scenario where the bean has to be changed.

Example14

35:Deep Dive of Autowiring inside Spring-Theory:

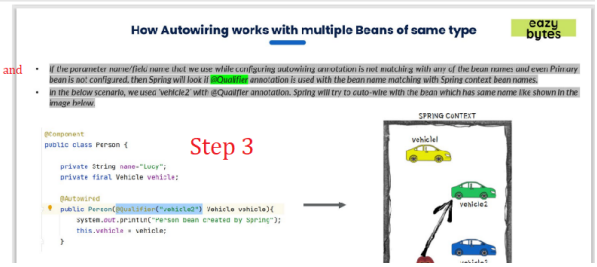


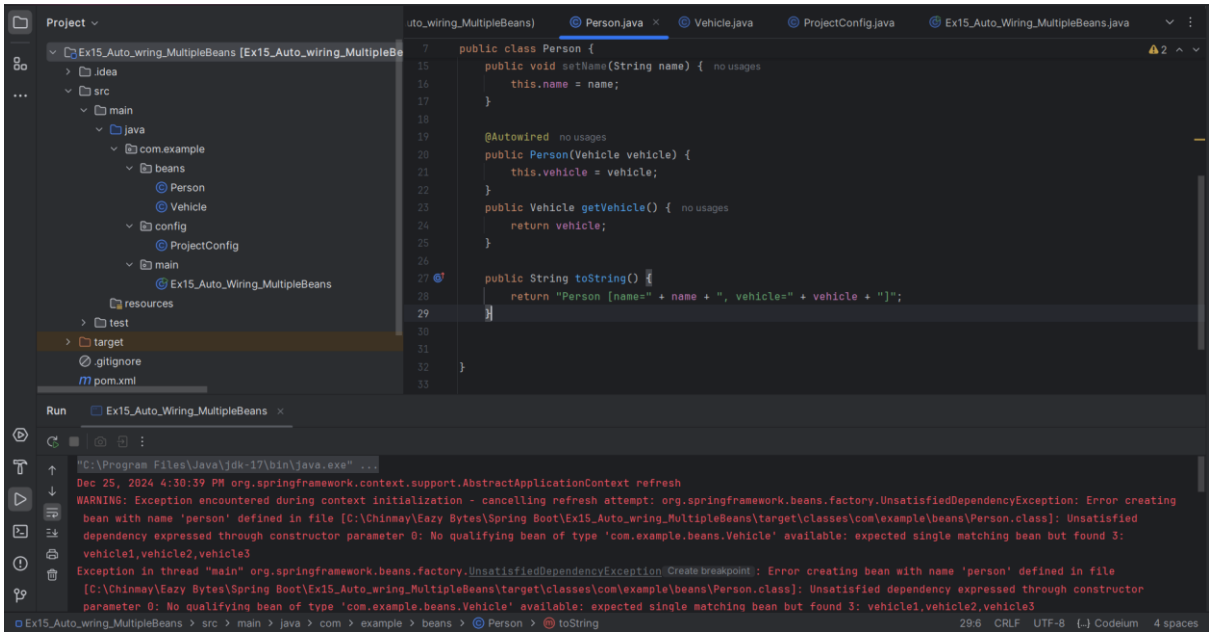


From Spring Version 6.1.0 the support for Step1 is removed .We can no more autowire the beans using their names.

If Spring is not able to resolve the which bean it has to injecct even after following the three steps then will raise and Exception

why can't you mention the same parameter name directly inside constructor parameter name like vehicle2.
The problem with that approaches in future. If another developer is working, definitely. They just think like this, just a parameter name and they can change based upon their requirements.So there is no readability, there is no control over it.
And if someone changes the parameter name, then definitely all your auto wiring and dependency injections will not work like you are expecting.But with the help of other rate qualifier annotation, we are clearly telling to the developers.
I'm trying to auto wire with the beam name vehicle2, so don't ever change the value that I have mentioned inside the qualifier annotation, but they are free to change the parameter name or field name inside the person class.

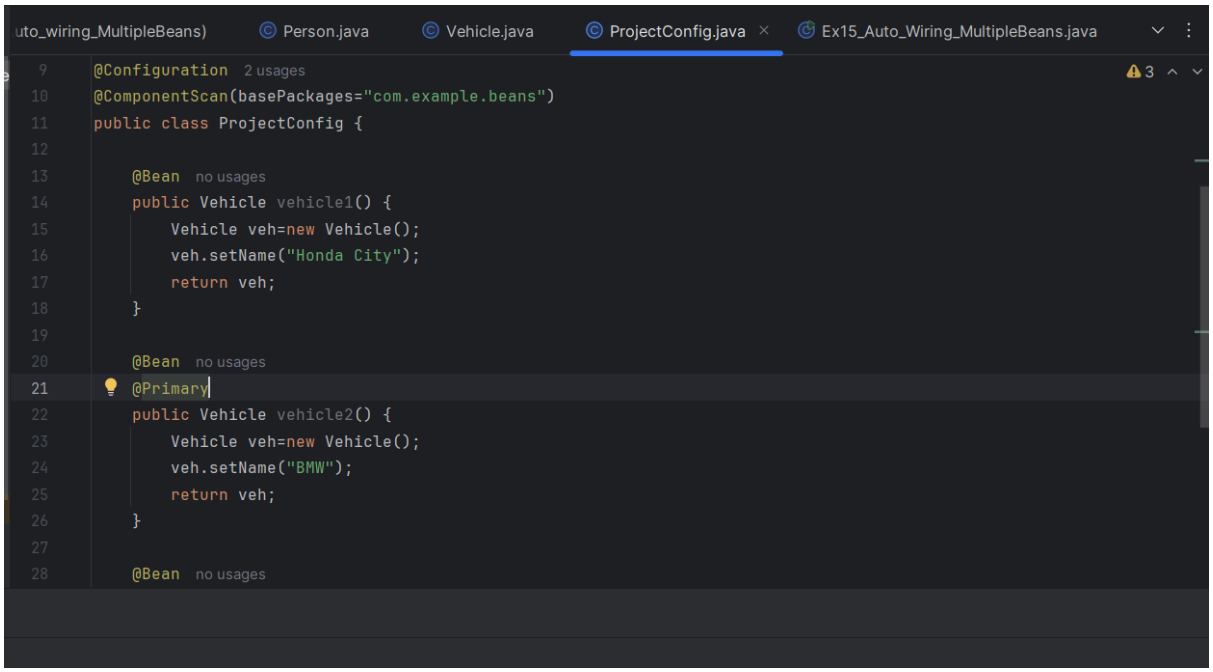




If don't specify the particular bean then it will lead unsatisfied bean Exception as there were multiple beans of same type.

For Spring version 6.1.0 we cannot autowire the beans using their names.

We can make one of the bean as Primary Bean:



In the case if we have multiple beans of same type and it is resulting to Ambiguity then we can one Bean as primary .So that bean will be invoked in case of ambigous scenario.

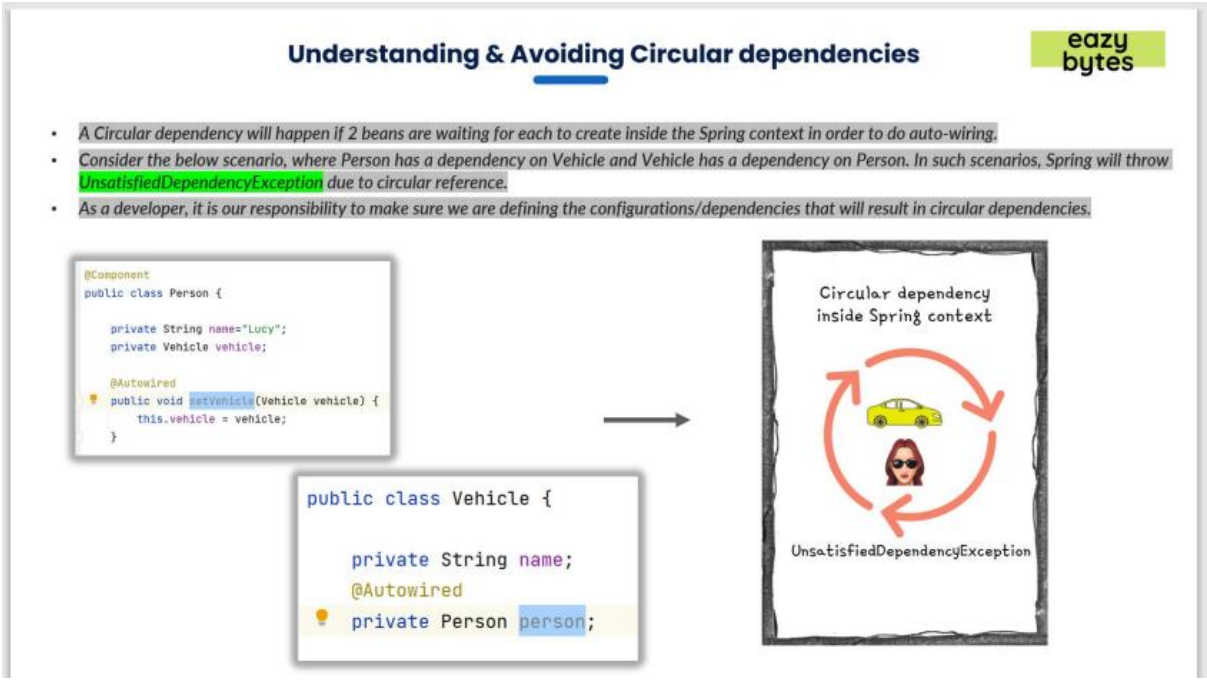
Another way:

```java
public class Person {
    }

    @Autowired  no usages
    public Person(@Qualifier("vehicle1") Vehicle vehicle) {
        this.vehicle = vehicle;
    }
    public Vehicle getVehicle() {  1 usage
        return vehicle;
    }

    public String toString() {
        return "Person [name=" + name + ", vehicle=" + vehicle + "]";
    }


}
```

We can user @Qualifier Annotation to pass the bean name in the Constructor .So the same will be taken from the context and will be injected .

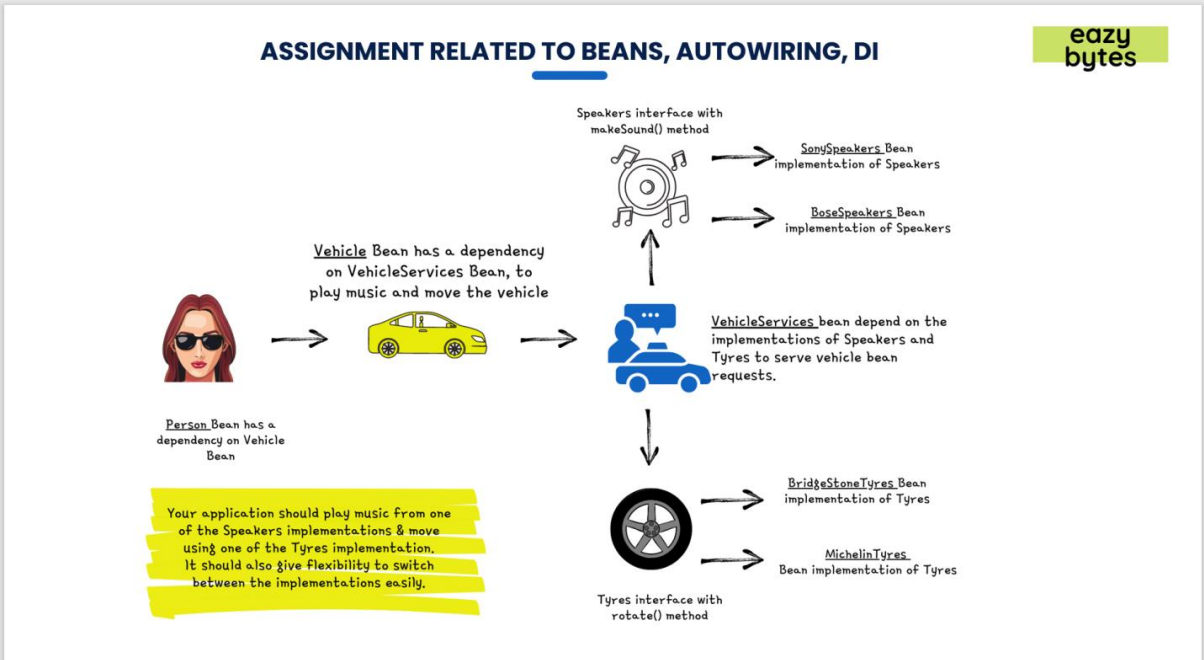
Example 15:

37.Understanding and Avoiding Circular dependencies



There is no way to avoid it we need to make sure that it will not happen.

38.Problem statement for Assignment related to Beans,AutoWiring,DI



On top of interface we will never use any kind of annotations like @Bean or @Component because we can't create beans from interfaces.we can only create from Java classes.

```
@Component("vehicleBean")  4 usages
public class Vehicle {

    private String name="Honda";  3 usages
    private final VehicleService vehicleService;  2 usages
```

Vehicle class bean will be named with vehicleBean.

```
@Component(value="personbean")  no usages
public class Person {
    Ctrl+L to Chat, Ctrl+I to Command
```

Person class bean will be named with personbean

```java
@Configuration  2 usages
@ComponentScan(basePackages = {"com.example.beans","com.example.implementation"})
@ComponentScan(basePackageClasses = {com.example.services.VehicleService.class})
public class ProjectConfig {
}
```

First approach

(@ComponentScan(basePackages{"com.example.beans","com.example.implementation"}))

The very first one has an advantages, like if your package has 100 classes, you don't have to mention all those 100 classes.

You just mention the package name.

But the disadvantage with that is there might be some classes inside the same package where you didn't mention any bean configurations.

Still, your spring IOC container will try to scan them and try to understand which is a kind of performance issue during startup.

Second Approach:

@ComponentScan(basePackageClasses = {com.example.services.VehicleService.class})

So to overcome that we can use another style which is by mentioning directly the class name. So when you are using class name style of configurations, definitely you have an advantage where in future if someone tries to change the class name, they will definitely get a compilation error because the class name directly mentioned here inside the component scan and the compilation will fail.

Whereas if someone tried to change the package name, since this is a string style of configurations, even if someone tried to change the package name, you won't get any compilation error.

So there can be a chance of mistake if someone is not aware of about this components scan package configurations. So that's why, based upon the scenario you are into and based upon the requirement that you have inside your application, you can feel free to choose any one of these approaches.

Ex:16