Spring Framework

https://www.youtube.com/watch?v=If1Lw4pLLEo

Introduction:

The benefits of using Spring framework: IoC, Dependency Injection, MVC,...
We will heavily rely on Maven to build our project

Dependency Injection:

Especially in the bug project, our class often depends on other class.
i.e. Laptop depends on Hardware components:

```
class Laptop {
HardDrive obj1;
RAM obj2;
...
}
```

These dependency will be used as object in runtime, which might change in the future.
i.e. version 1 is

```
HardDrive obj1 = new HitachiHD()
```

and version 2 become

```
HardDrive obj1 = new SamsungHD()
```

Of course, we can change this hardcode, update the application whenever your dependency object change.
Or we can rely on the Dependency Injection. In Spring Framework, by injecting the HardDrive dependency and define the injection target in another file. The Spring Container will create the object and inject into our Laptop class.
By doing this, we only need to update the file when the dependency object is changed.
i.e.

```
@Component
class Hitachi implements HardDrive {
...
}

Class Laptop {
@Autowired
HardDrive obj;
}
```

Conclusion: Benefits of Dependency Injection
(1) Easy to manage, less hard code in your JAVA code
(2) Easy to test. Assume your dependency class are tested, we don't need to repeat unit testing this dependency. Just mock this object in your class unit test.

Maven:

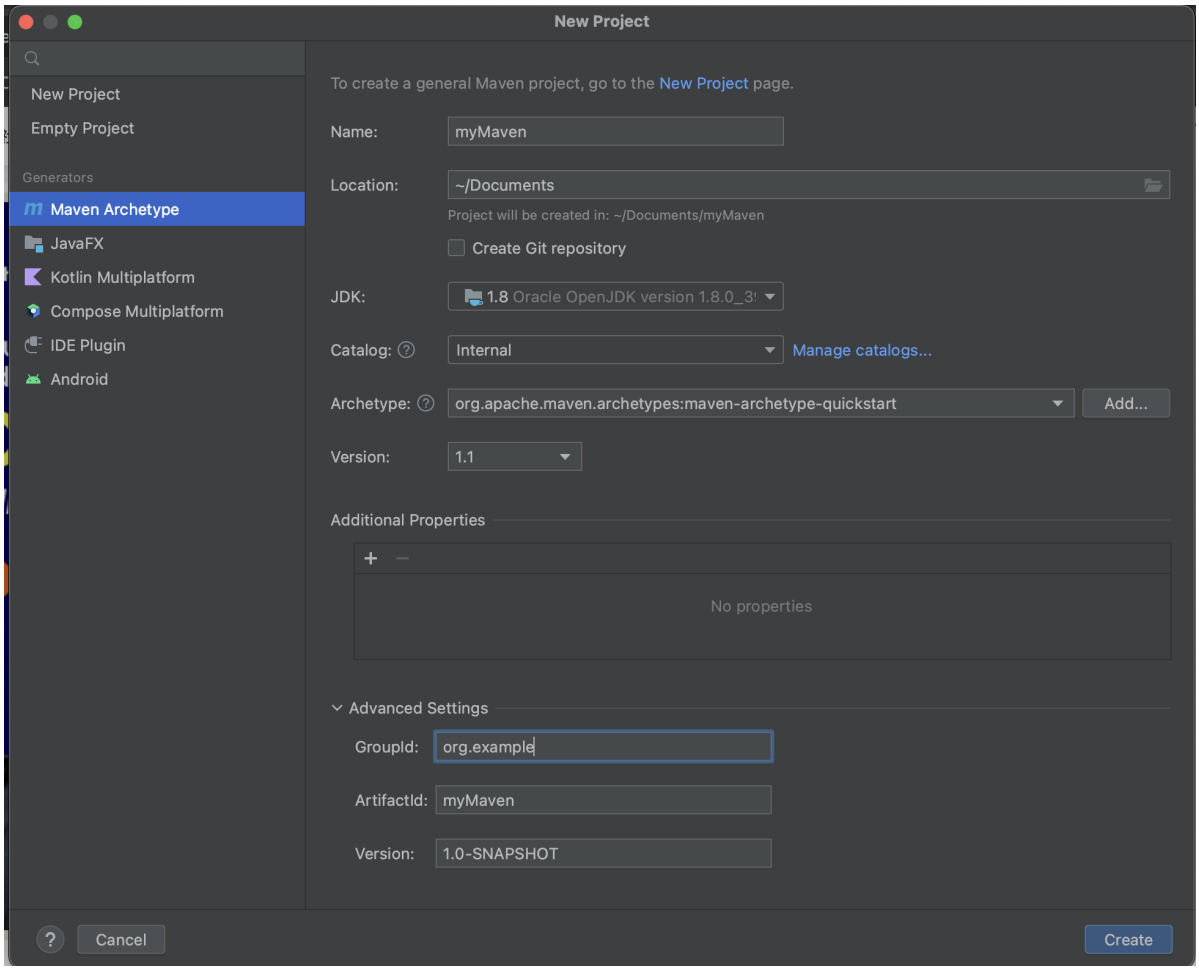
Before Maven, you need manually download EVERY jar file into your project (Go to Junit website for Junit. Go to spring website for SpringFramework, etc...). Maven is central repository to manage this. All you need to do is declare your dependency in the XML file, Maven will download it for you.
There is remote repository and your local repository, caching hierarchy.
Maven use a pom.xml file, all the dependency libraries will be in

```
<dependencies>
<dependency>
...
</dependency>
</dependencies>
```

The dependency library will follow the naming routine: Project->GroupID->ArtifactID->Package
Maven is not only building tool, you can also use Gradle (common used in Android development).

Maven Demo:

We can rely on IntelliJ to create a Maven project. Choose "quickstart" since we are just creating a simple java application

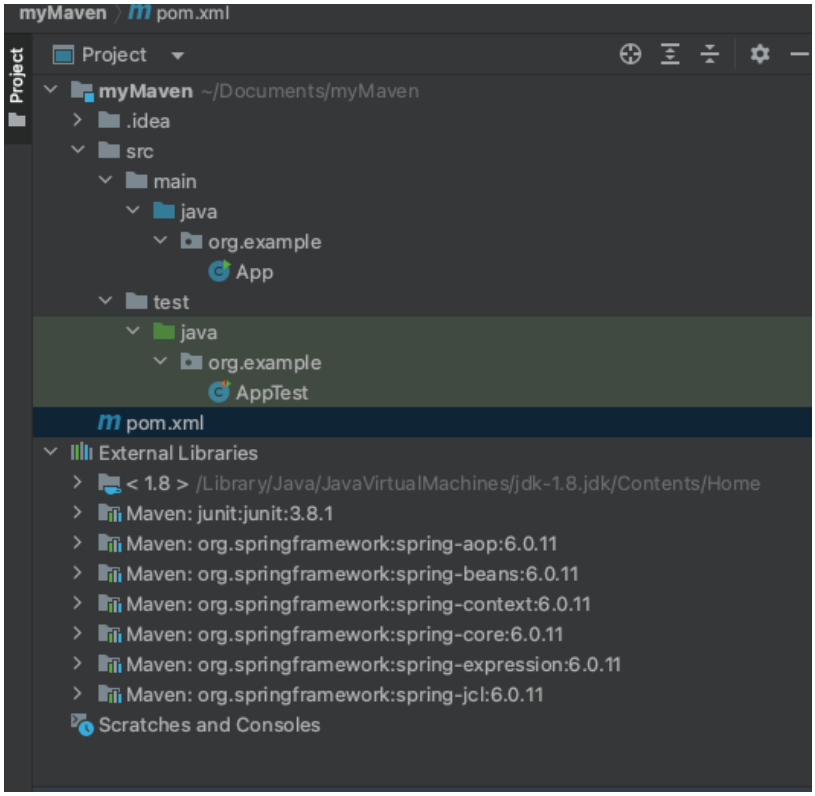


In the pom.xml file

Add springframework into dependencies section (the embedded code can be download from spring website)

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>6.0.11</version>
</dependency>
```

and reload the Maven, you will see the libraries are pulled into our project



Application Context:

Example:

Initially, we are calling the first dependency class from our main JAVA application:

```
public class Car {
public void drive() {
System.out.println("Driving a car");
}
}

public class App {
public static void main( String[] args ) {
Car car = new Car();
car.drive();
}
}
```

Later on, we need to update to another dependency class

```
public class Bike {
public void drive(){
System.out.println("Riding a bike");
```

```
}
}
public class App {
public static void main( String[] args ) {
Bike bike = new Bike();
bike.drive();
}
}
```

We can keep doing this whenever we need to update the new dependency.

However, with dependency injection, we can define a common Interface "Vehicle"

```
public interface Vehicle {
void drive();
}

public class Bike implements Vehicle {
public void drive(){
System.out.println("Riding a bike");
}
}
public class Car implements Vehicle {
public void drive(){
System.out.println("Driving a car");
}
}
```

In the main, there are two ways to inject the dependency object (getBean()).
(1) BeanFactory (org.springframework.beans.factory.BeanFactory): Usually used in small application.
(2) ApplicationContext (org.springframework.context.ApplicationContext): Usually used in web application.
ApplicationContext is Superset of BeanFactory.

```
public class App {
public static void main( String[] args ) {
ApplicationContext context = new ClassPathXmlApplicationContext();
Vehicle obj = (Vehicle) context.getBean("vehicle");
// getBean() will give you the obj
obj.drive();
}
}
```

However, you will have the following error. Because "vehicle" is an interface, the ApplicationContext still don't know which class it should create the object (is it Car or Bike)?

We need to define which object we need when we are trying to get a "vehicle". This file is xml

```
// spring.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="vehicle" class="org.example.Car"></bean>

</beans>
```

and the ApplicationContext will rely on this xml file

```
ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
Vehicle obj = (Vehicle) context.getBean("vehicle");
obj.drive();
```

From now on, if you want to update the Vehicle to Bike, just update the xml file. No need to update JAVA code the recompile.

Annotation Based configuration:

In previous sample, we use xml file to define bean. Based on the bean definition, user ApplicationConext to create object and inject into our class.
Which is (1) xml-based configuration.
Here is two other ways to config the bean: (2) Annotation-based configuration (3) JAVA configuration
In xml-based configuration, you will need to keep updating the xml file

```
<bean id="vehicle" class="org.example.Car"></bean>
->
<bean id="vehicle" class="org.example.Bike"></bean>
```

Using a Bean annotation, you can add @Components annotation to the class. It will auto generate a bean ID based on class name.

```
@Component
// bean id: "bike"
public class Bike implements Vehicle {...}

@Component
// bean id: "car"
public class Car implements Vehicle {...}
```

But the xml still need to know where to look for beans. Add component-scan to search the package.

```
<context:component-scan base-package="org.example2"></context:component-scan>
```

which will scan all components under "org.example2" and match the bean ID.

Final, just use the bean ID to inject the object.

```
public class AppExample2 {
public static void main( String[] args ) {
ApplicationContext context = new
FileSystemXmlApplicationContext("src/main/java/spring2.xml");
Vehicle obj = (Vehicle) context.getBean("car");
obj.drive();
}
}
```

Bean property:

You can add property tag inside your bean tag and initialize the object created by Spring. In another word, Spring can dynamic create the objects.
This is the object you are injecting.

```
public class Tire {
private String brand;
public Tire(String brand) { this.brand = brand; } // contructor

public void setBrand(String brand) { this.brand = brand; } // Setter

public String getBrand() { return brand; } // Getter
}
```

Adding property tag, with initial value (Use xml-based configuration as example here, initial value as "FireStone").

```
<bean id="tire" class="org.example2.Tire">
<property name="brand" value="FireStone"></property>
</bean>
```

Inject into our main application

```
public class AppExample2 {
public static void main( String[] args ) {
Tire obj3 = (Tire) context.getBean("tire");
System.out.println(obj3.getBrand()); // FireStone
}
}
```

Later on, just update the xml file to dynamic update the attribute of object you are getting.

Construction Injection:

In previous sample, we use the Setter Injection. Essentially the <property> tag in bean will call the Setter method in target class.
(<property name="brand" value="FireStone"> tag is calling setBrand("FireStone"))

You can also rely on the Constructor Injection to decide the initial value of the objects. The <constructor-arg> will call the constructor method in target class. (constructor-arg value="Dunlop"> tag is calling Tire("Dunlop"))

```
<bean id="tire" class="org.example2.Tire">
<constructor-arg value="Dunlop"></constructor-arg>
</bean>
```

AutoWired Annotation:

with @Autowire, whenever we need object, we will go back to xml file to search for bean ID

We already use Annotation-base configuration to inject the Vehicle (Car/ Bike) and the xml-base configuration to inject the Tire. Directly calling works like a charm:

```
// spring3.xml
<context:component-scan base-package="org.example3"></context:component-scan>
<bean id="tire" class="org.example3.Tire">
<property name="brand" value="FireStone"></property>
</bean>
```

```
@Component
public class Car implements Vehicle {
public void drive(){
System.out.println("Driving a car Example3");
}
}
@Component
public class Bike implements Vehicle {
public void drive(){
System.out.println("Riding a bike Example3");
}
}
```

```
public class AppExample3
{
public static void main( String[] args )
{
System.out.println( "Hello World App from Example3!" );
ApplicationContext context = new
FileSystemXmlApplicationContext("src/main/java/spring3.xml");
Vehicle obj = (Vehicle) context.getBean("car");
obj.drive();
Vehicle obj2 = (Vehicle) context.getBean("bike");
obj2.drive();

Tire t = (Tire) context.getBean("tire");
System.out.println(t.getBrand());
}
}
->
Driving a car Example3
Riding a bike Example3
FireStone
```

However, consider following scenario: The Vehicle depends on the Tire. Above dependency injection won't work:

```
@Component
public class Car implements Vehicle {
private Tire tire;
public Tire getTire() { return tire; }
public void setTire(Tire tire) { this.tire = tire; }
public void drive(){
System.out.println("Driving a car Example3; Tire: " + tire);
}
}

@Component
public class Bike implements Vehicle {
private Tire tire;
public Tire getTire() { return tire; }
public void setTire(Tire tire) { this.tire = tire; }
public void drive(){
System.out.println("Riding a bike Example3; Tire: " + tire);
}
}

->
Driving a car Example3; Tire: null
Riding a bike Example3; Tire: null
FireStone
```

The Car.tire and Bike.tire are both null. For Car/Bike, the tire is a property of this class. Of course we can do nested-xml-based configuration, like

```
// spring3.xml
<bean id="vehicle" class="org.example3.Car">
<property name="tire" ref="tire"/>
</bean>
<bean id="tire" class="org.example3.Tire">
<property name="brand" value="FireStone"></property>
</bean>
```

Use Setter Injection to inject Tire object to Vehicle object.

```
Vehicle obj = (Vehicle) context.getBean("vehicle");
obj.drive();
Tire t = (Tire) context.getBean("tire");
System.out.println(t.getBrand());

->
Driving a car Example3; Tire: FireStone
FireStone
```

But we have better solution: AutoWired. Simply add @Autowired for the object you want to get. Spring will search xml file and look for Bean for injection.

```
@Component
public class Car implements Vehicle {
@Autowired
private Tire tire;
...

@Component
public class Bike implements Vehicle {
@Autowired
private Tire tire;
...
```

where Bean "tire" already in xml file, with Setter Injection.

```
<bean id="tire" class="org.example3.Tire">
<property name="brand" value="FireStone"></property>
</bean>
```

```
Vehicle obj = (Vehicle) context.getBean("car");
obj.drive();
Vehicle obj2 = (Vehicle) context.getBean("bike");
obj2.drive();

Tire t = (Tire) context.getBean("tire");
System.out.println(t.getBrand());

->
Driving a car Example3; Tire: FireStone
Riding a bike Example3; Tire: FireStone
FireStone
```

You can even delete the xml for tire Bean. Using @Component to auto generate tire Bean.

```
<!-- <bean id="tire" class="org.example3.Tire">-->
<!-- <property name="brand" value="FireStone"></property>-->
<!-- </bean>-->

@Component
public class Tire {
```

```
...
```

But this will lose the Setter Injection.

Annotation Configuration Bean:

Check another example:

```
public class Samsung {
public void config(){
System.out.println("Samsung Class");
}
}

public class AppExample4_Annotation {
public static void main( String[] args ) {
System.out.println( "Hello World App from Example4!" );
// Asking factory to give us the object
ApplicationContext factory = new AnnotationConfigApplicationContext();

Samsung s7 = factory.getBean(Samsung.class);
s7.config();
}
}
```

This will not work because there is no any Bean for Spring knowing which class
In xml-based, we can use

```
<bean id="phone" class="org.example4.Samsung"></bean>-->
```

We don't need that in Annotation-based configuration.
Creating a AppConfig.java class

```
@Configuration
public class AppConfig {
@Bean
public Samsung getPhone(){
return new Samsung();
}
}
```

Instead of fetching the object from xml Bean, we are fetching the object from the definition in AppConfig. With @Configuration, Spring will look for Bean reference in

this AppConfig. We also add the reference to AppConfig to AnnotationConfigApplicationContext()

```
ApplicationContext factory = new
AnnotationConfigApplicationContext(AppConfig.class);
Samsung s7 = factory.getBean(Samsung.class);
s7.config();

->
Samsung Class
```

Continue to add dependency to Samsung. Assume we need CPU for the phone:

```
public interface MobileCPU {
void process();
}
public class Intel implements MobileCPU{
public void process() { System.out.println("Intel CPU"); }
}
```

Add Bean in AppConfig

```
@Bean
// notice the getter function doesn't matter. the bean id is auto generated based on
class name
public MobileCPU getCPU(){
return new Intel();
}
```

Finally we can add CPU into our phone class. Use @Autowired annotation so Spring knows to check AppConfig for injection target. (Recall: we use @Autowired before, but last time Spring was checking xml for Bean)

```
public class Samsung {
@Autowired
// will check the AppConfig
MobileCPU cpu;

public MobileCPU getCup() {
return cpu;
}

public void setCup(MobileCPU cup) {
this.cpu = cup;
}
```

```
public void config(){
System.out.println("Samsung Class");
cpu.process();
}
}

->
Samsung Class
Intel CPU
```

We done everything without XML. But we can do more. Lots of Bean annotation in AppConfig seems redundant.

Annotation Components: AutoWired Primary Qualifier:

Remove the everything with Bean annotation in AppConfig will trigger BeanNotFound Error.
We need to Add @Component to the Samsung and Intel. (Recall: @Component will auto generate Bean for class)

```
@Component
public class Samsung {
...

@Component
public class Intel implements MobileCPU {
...
```

Adding @ComponentScan to AppConfig and remove all Bean decoration

```
@Configuration
@ComponentScan(basePackages = "org.example5") // Means all components are pre-
defined
public class AppConfig {
// Removing everything, we will get Bean-not-found error
// Adding @Component to Samsung and Intel
}

->
Samsung Class example 5
Intel CPU example5
```

By default, the @Component annotation will create a Bean ID = non-qualified and de-capitalize class name.
Take following as an example:
if we have multiple CPU:

```
@Component
public class Amd implements MobileCPU {
public void process() { System.out.println("AMD CPU example5"); }
}

@Component
public class Intel implements MobileCPU {
public void process() { System.out.println("Intel CPU example5"); }
}
```

Now we have two Beans represent MobileCPU. Which one should we inject?
When there is conflict, the Bean with @Primary annotation will be chosen.

```
@Component
@Primary
public class Intel implements MobileCPU {
public void process() { System.out.println("Intel CPU example5"); }
}
->
Samsung Class example 5
Intel CPU example5
```

```
@Component
@Primary
public class Amd implements MobileCPU {
public void process() { System.out.println("AMD CPU example5"); }
}
->
Samsung Class example 5
AMD CPU example5
```

Or you will Add @Qualifier("") annotation to choose which Bean you want to inject

```
@Component
public class Samsung {
@Autowired
```

```
@Qualifier("amd")
MobileCPU cpu;
...

->
Samsung Class example 5
AMD CPU example5
```