Introduction to Spring Framework

By

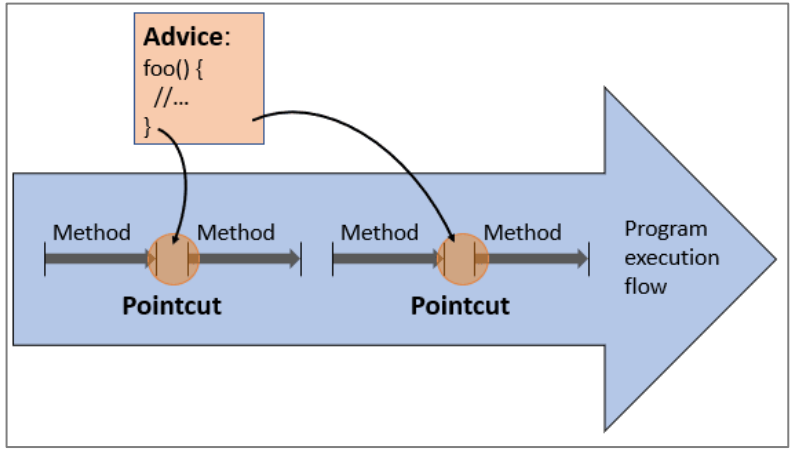
Pichet Limvajiranan

Container Services

- When we talk about containers, it is expected that any container should be capable of providing several basic services to components managed in its environment.
  - Life-cycle management
  - Dependency resolution
  - Component lookup
  - Application configuration
- In addition to those features, it will be very useful if the container is able to provide following middleware services:
  - Transaction management
  - Security
  - Thread management
  - Object and resource pooling
  - Remote access for components

AOP Concepts

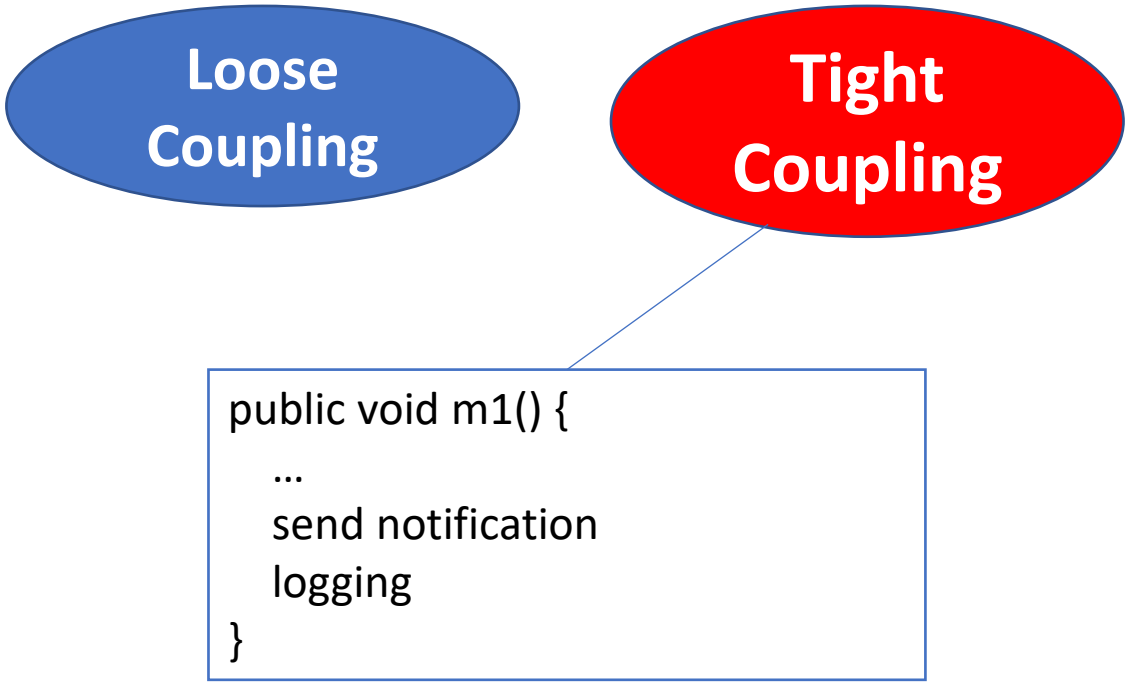
- Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.
- AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns.
- A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.



Why use AOP? (1)

- It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

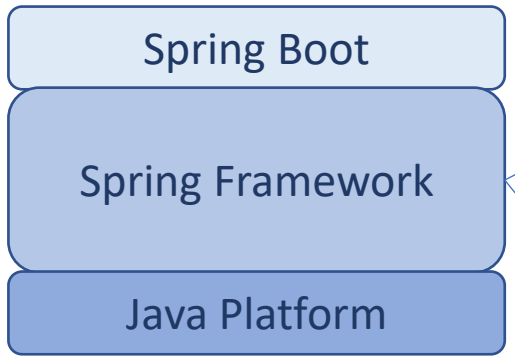
```
class A {
        public void m1(){...}
        public void m2(){...}
        public void m3(){...}
        public void m4(){...}
        public void m5(){...}

        public void n1(){...}
        public void n2(){...}

        public void p1(){...}
        public void p2(){...}
        public void p3(){...}
}
```

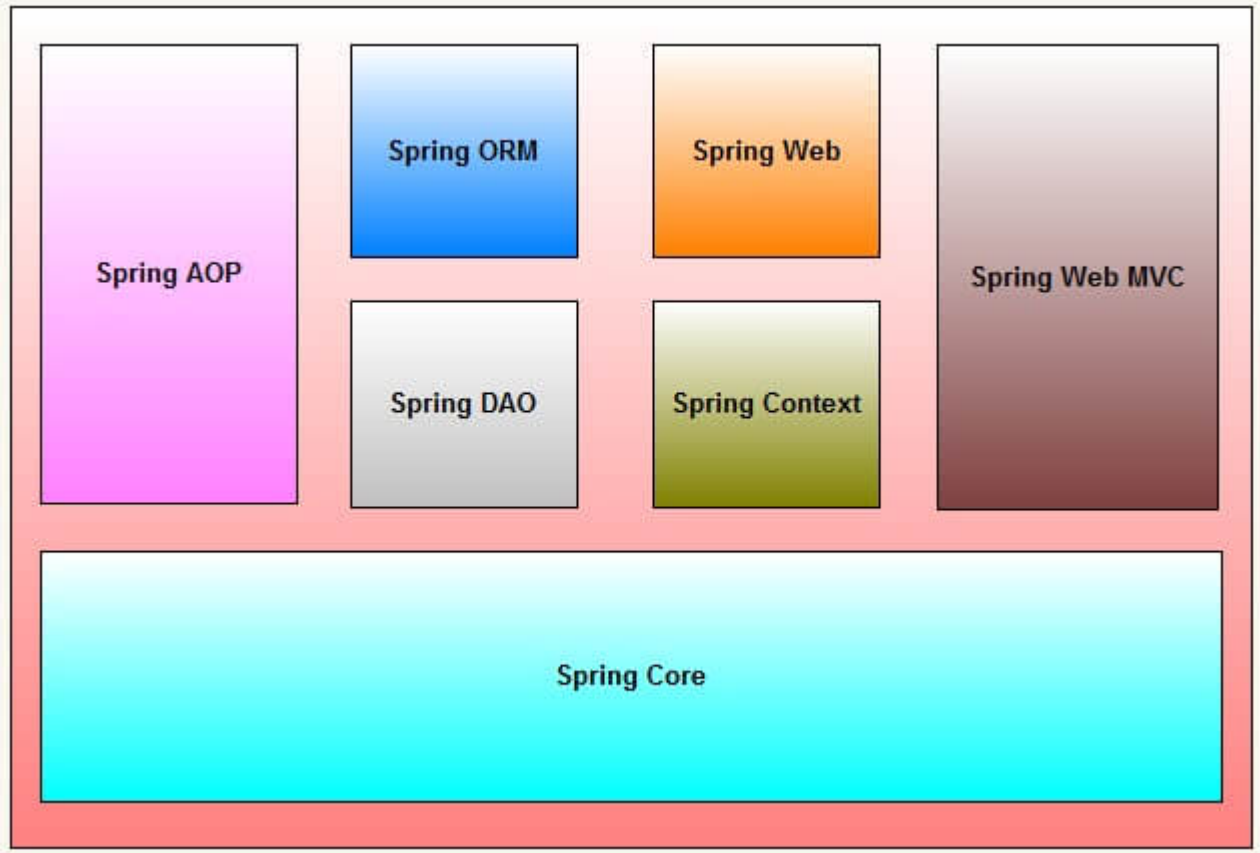


Why use AOP? (2)

- There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.
- Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.
- Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.
- But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.
- **Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.**
- In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

What is Spring Boot?





- Spring Boot is a project that is built on the top of the Spring Framework.
- It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

Spring Boot vs Spring Framework

- Spring: Spring Framework is the most popular application development framework of Java. The main feature of the Spring Framework is **dependency Injection or Inversion of Control (DI or IoC)**. With the help of Spring Framework, we can **develop a loosely coupled application**.
- Spring Boot: Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with **minimal or zero configurations**. It is better to use if we want to develop a simple Spring-based application or RESTful services.
- Spring framework is something that we need to **manually change almost all configurations**, while Spring Boot already provides us with **zero or limited configuration** standalone applications.

Spring Framework Features (1)

Spring framework is an open source framework created to solve the complexity of enterprise application development.

Features:

- Lightweight
  - Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.
- **Inversion of control (IoC) / Dependency Injection (DI)**
  - The basic concept of the Dependency Injection or Inversion of Control is that, programmer do not need to create the objects, instead just **describe how it should be created**.
  - No need to directly connect your components and services together in program, instead just **describe which services are needed by which components in a configuration file/xml file**. The Spring IOC container is then responsible for binding it all up.

Spring Framework Features (2)

- Aspect oriented (AOP)
  - Spring supports Aspect oriented programming.
  - Aspect oriented programming refers to the programming paradigm which isolates secondary or supporting functions from the main program's business logic.
  - AOP is a promising technology for separating crosscutting concerns, something usually hard to do in object-oriented programming. The application's modularity is increased in that way and its maintenance becomes significantly easier.
- Container
  - Spring contains and manages the life cycle and configuration of application objects.

Spring Framework Features (3)

- MVC Framework
  - Spring comes with MVC web application framework, built on core Spring functionality.
  - This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.
- Transaction Management
  - Spring framework provides a generic abstraction layer for transaction management. This allowing the developer to add the pluggable transaction managers, and making it easy to demarcate transactions without dealing with low-level issues.
- JDBC Exception Handling
  - The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy.
  - Spring provides best Integration services with Hibernate, JDO and iBATIS (MyBatis)

Spring Framework Modules (1)

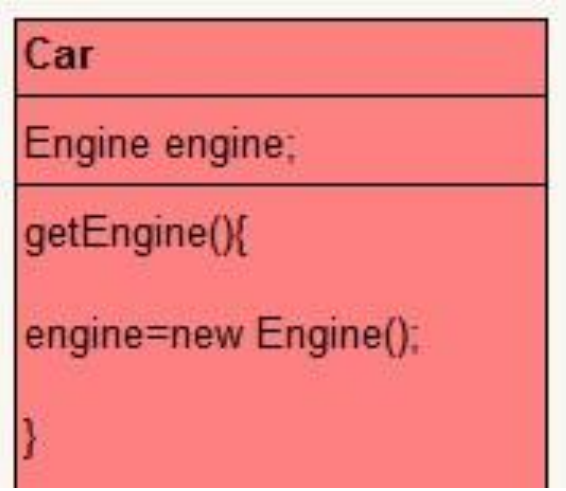
- Modules in the Spring framework are:
- Spring AOP
  - One of the key components of Spring is the AOP framework.
  - AOP is used in Spring:
    - To provide declarative enterprise services.
    - The most important such service is declarative transaction management, which builds on Spring's transaction abstraction.
    - To allow users to implement custom aspects, complementing their use of OOP with AOP.
- Spring ORM
  - The ORM package is related to the database access. It provides integration layers for popular object-relational mapping APIs, including JDO, Hibernate and iBatis.
- Spring Web
  - The Spring Web module is part of Spring's web application development stack, which includes Spring MVC.

Spring Framework Modules (2)

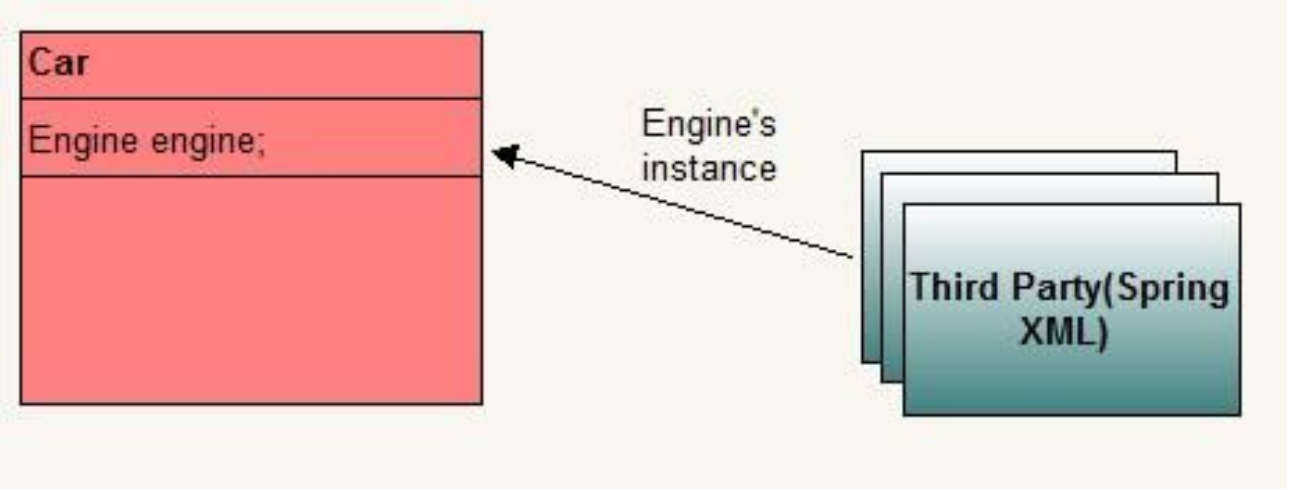
- Spring DAO
  - The DAO (Data Access Object) support in Spring is primarily for standardizing the data access work using the technologies like JDBC, Hibernate or JDO.
- Spring Context
  - This package builds on the beans package to add support for message sources and for the Observer design pattern, and the ability for application objects to obtain resources using a consistent API.
- Spring Web MVC
  - This is the Module which provides the MVC implementations for the web applications.
- Spring Core
  - The Core package is the most import component of the Spring Framework.
  - This component provides the Dependency Injection features.
  - The BeanFactory provides a factory pattern which separates the dependencies like initialization, creation and access of the objects from your actual program logic.

Spring Inversion of Control (IoC)

- You do not create your objects but describe how they should be created.
- You don’t directly connect your components and services together in code but describe which services are needed by which components in a configuration file.
- An IOC container is then responsible for hooking it all up.



Without DI



With DI

Types of Dependency Injection

- Setter Injection:Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.
- Constructor Injection:Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator.
- Interface Injection: In interface-based dependency injection, we will have an interface and on implementing it we will get the instance injected.

Constructor Injection

applicationContext.xml

```xml
<bean id="addr" class="sit.int204.example.model.Address">
    <constructor-arg value="Thungkru"></constructor-arg>
    <constructor-arg value="Bangkok"></constructor-arg>
    <constructor-arg value="Thailand"></constructor-arg>
</bean>
```

```xml
<bean id="e" class="sit.int204.example.model.Employee">
    <constructor-arg value="10" type="int"></constructor-arg>
    <constructor-arg value="Somchai"></constructor-arg>
    <constructor-arg>
        <ref bean="addr"/>
    </constructor-arg>
</bean>
```

```java
public class Employee {
    private int id;
    private String name;
    private Address address;//Aggregation

    public Employee(int id, String name,
         Address address) {
        super();
        this.id = id;
        this.name = name;
        this.address = address;
    }
```

```java
public class Test {
    public static void main(String[] args) {
        ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");
        Employee employee = (Employee) context.getBean("e");
        employee.show();
```

Setter Injection

```xml
<bean id="addr2" class="sit.int204.example.model.Address">
    <property name="city" value="Banpong"></property>
    <property name="state" value="Ratchaburi"></property>
    <property name="country" value="Thailand"></property>
</bean>
```

```xml
<bean id="e2" class="sit.int204.example.model.Employee">
    <property name="id" value="101"></property>
    <property name="name" value="Somsri"></property>
    <property name="address" ref="addr2"></property>
</bean>
```

```java
public class Employee {
    private int id;
    private String name;
    private Address address;//Aggregation
    public void setId(int id) {
        this.id = id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");
        Employee employee = (Employee) context.getBean("e2");
        employee.show();
```

Spring Configurations

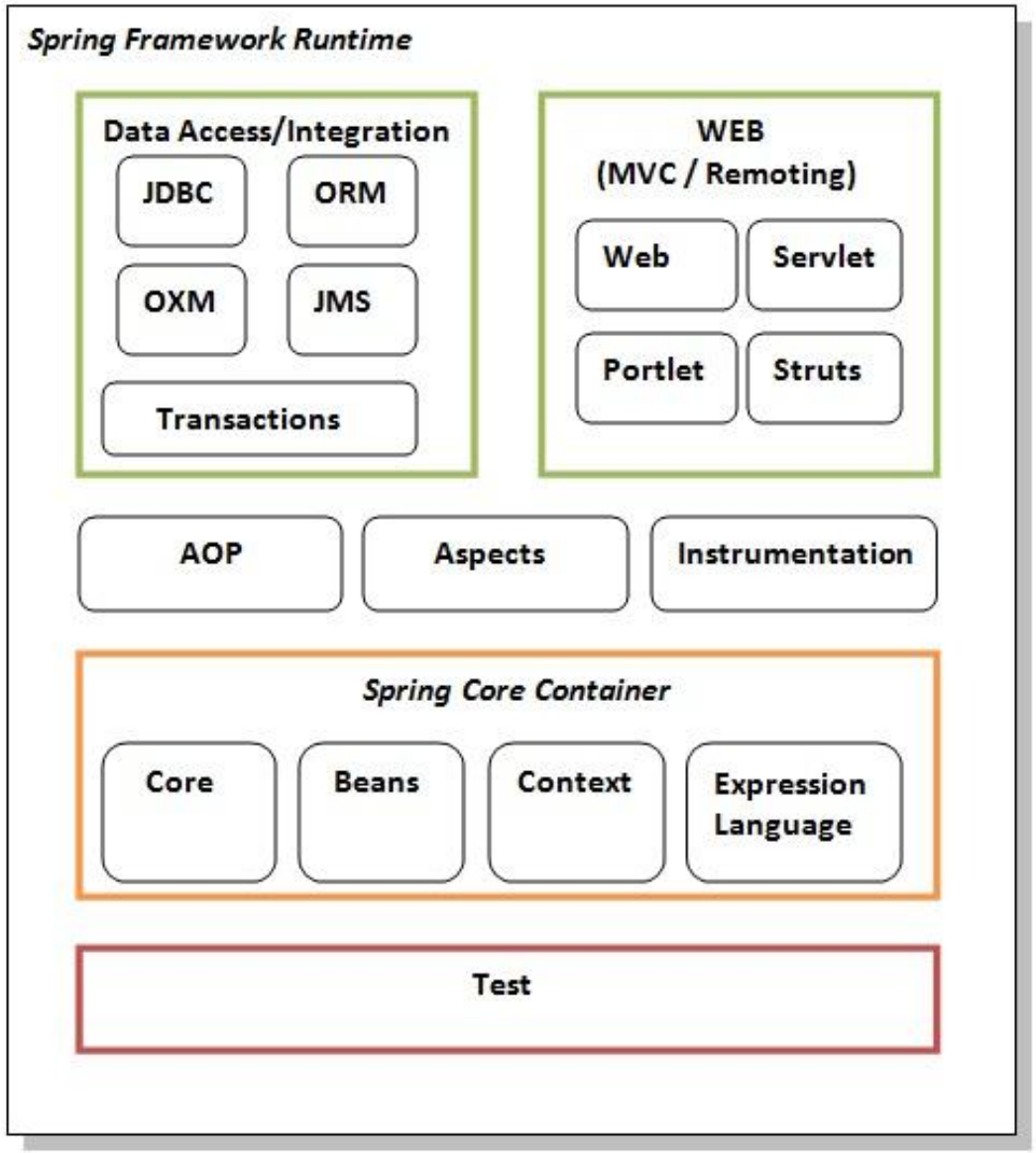
Spring Java based configuration

```java
@Configuration
public class ApplicationConfiguration {
    @Bean(name = "studentObj")
    public Student getStudent() {
        return new Student("Somchai Lim");
    }
}
```

Spring XML based configuration

```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-3.0.xsd">

    <bean id="studentbean" class="sit.int204.example.model.Student">
        <property name="name" value="Somchai Wong"></property>
    </bean>
```

Spring Framework Runtime



OOD: SOLID

https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

SOLID is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin (also known as Uncle Bob).

- S - Single-responsiblity Principle
- O - Open-closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Tight Coupling vs Loose Coupling

```
public class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine ;
    }

    public void start() {
        engine.turnOn();
    }
}
```

```
public class Car {
    private Engine engine;

    public Car() {
      this.engine = new Engine();
    }

    public void start() {
      engine.turnOn();
    }
}
```

Open-Closed Principle and Injection

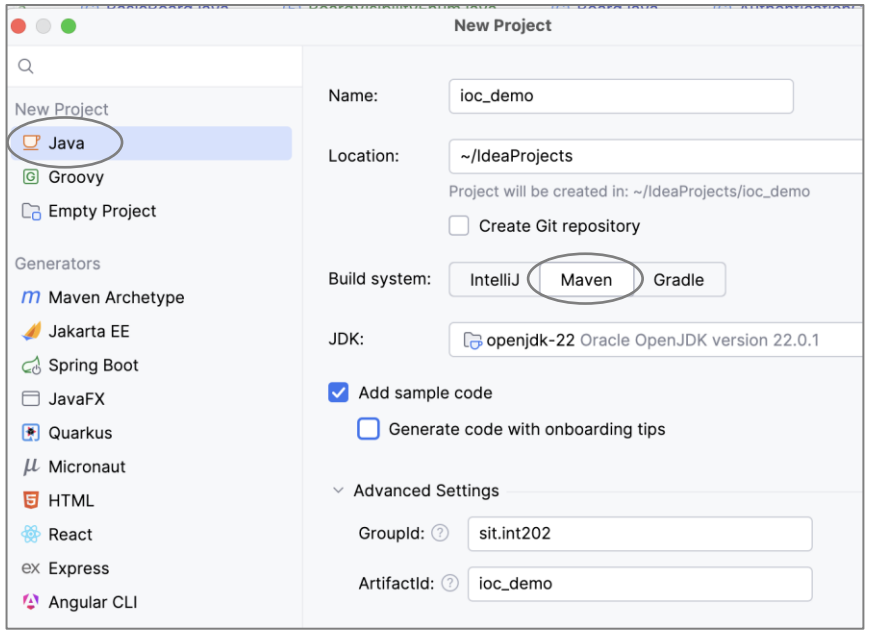
```
public class DieselEngine implements Engine {
  @Override
  public void turnOn() {
  }

  @Override
  public void accelerate() {
  }

  @Override
  public void turnOff() {
  }
}
```

```
public class GasolineEngine implements Engine {
  @Override
  public void turnOn() {
  }

  @Override
  public void accelerate() {
  }

  @Override
  public void turnOff() {
  }
}
```

```
Engine de = new DieselEngine();
Engine ge = new GasolineEngine();
Car car = new Car(de);  or  Car car = new Car(ge);
```

Spring Dependency Injection (IoC) Exercises

1. Create Maven Project
   - Name : ioc_demo
   - Group Id : sit.int204
   - Artifact Id : ioc_demo
   - Version : 1.0-SNAPSHOT
2. Add 3 Spring Dependency
   - Spring Core
   - Spring Context
   - Spring Beans



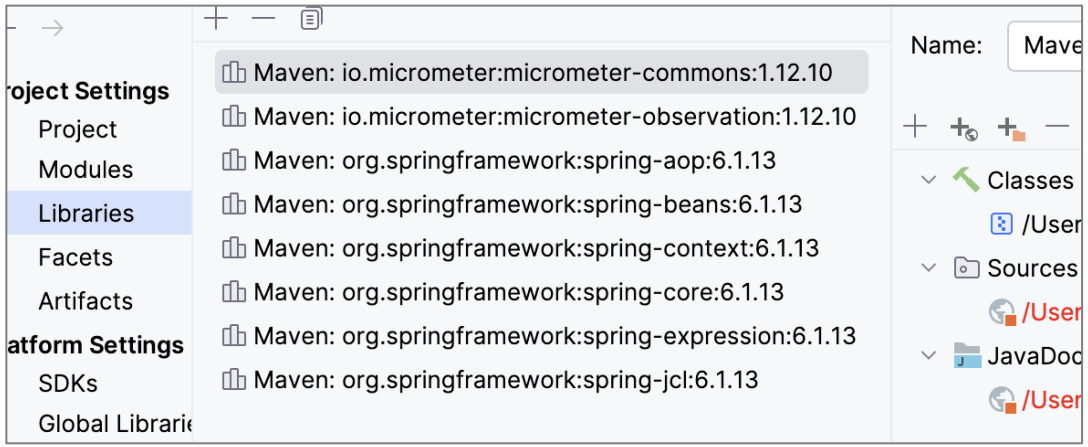

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>6.1.13</version>
</dependency>
```

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>6.1.13</version>
</dependency>
```

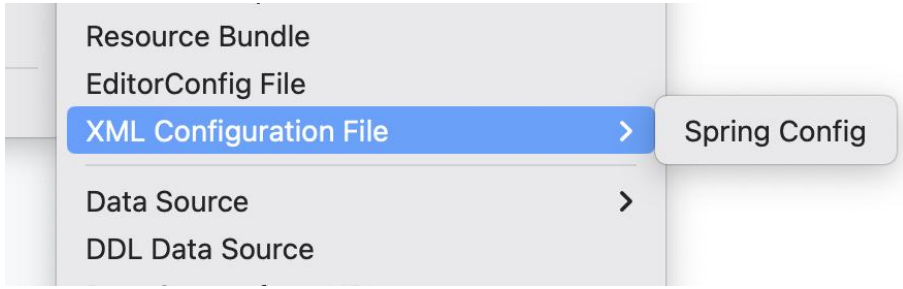
```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-beans</artifactId>
   <version>6.1.13</version>
</dependency>
```

3) Checking Spring Dependency

Menu: File -> Project Structures … (after reload project pom.xml)

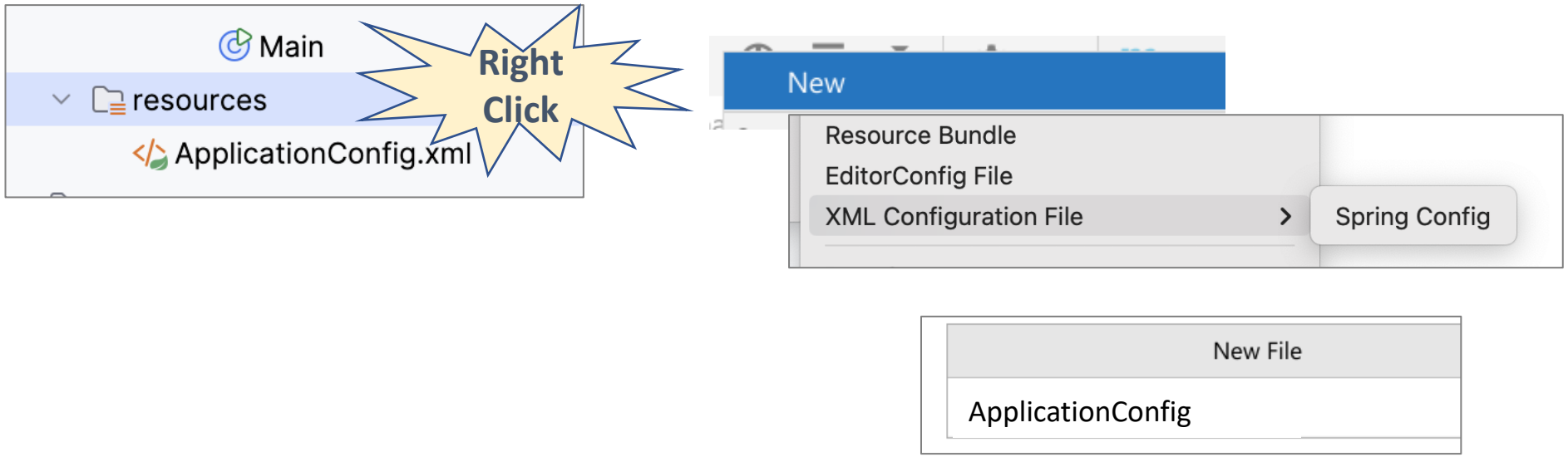


Sub-Menu item “XML Config File -> Spring Config” will be added to menu File->New



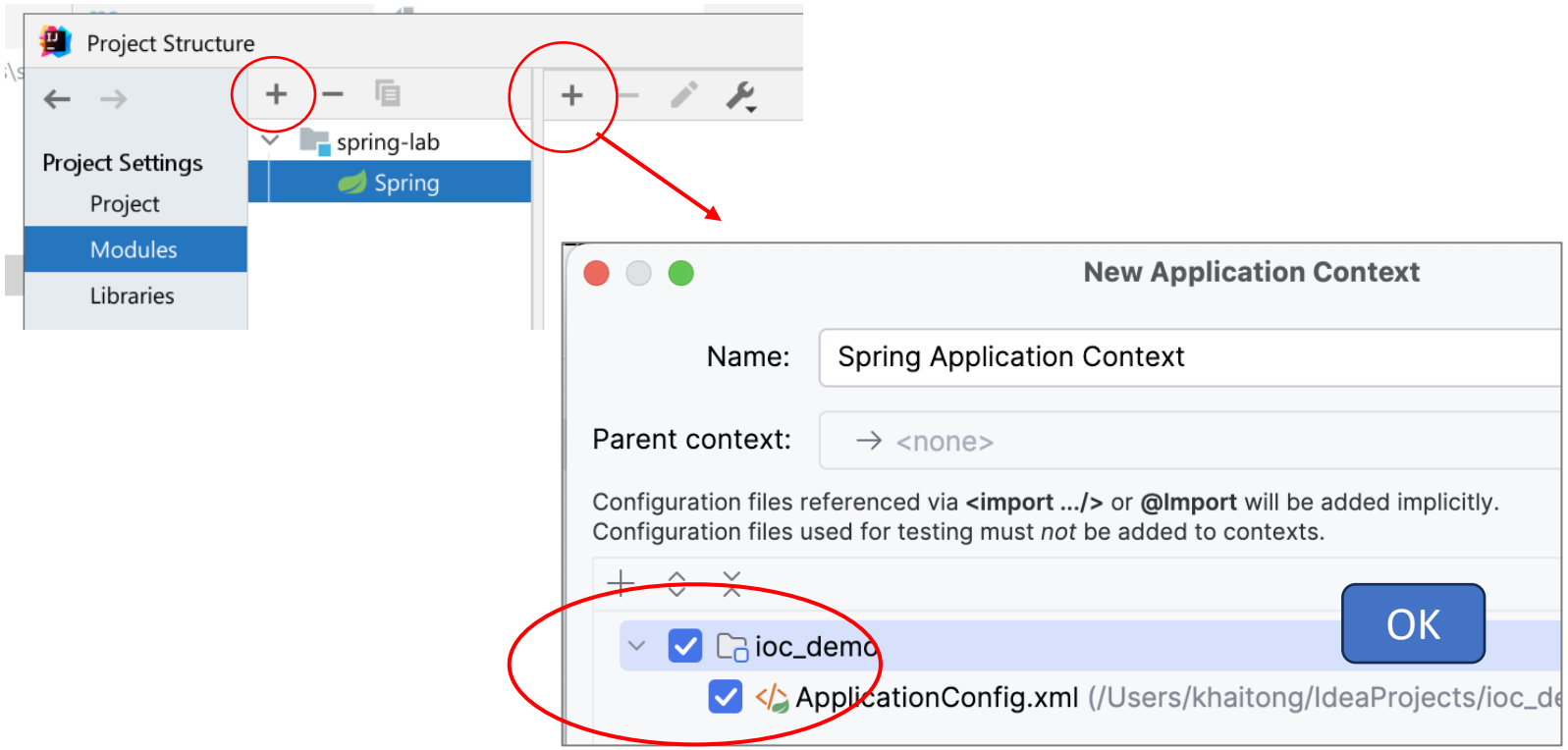
3) Create xml configuration file

1) Right Click on folder “resources”
2) Select New ->XML Configuration->Spring Config
3) Enter file name : ApplicationContext



Add Spring Application Context

Menu: File -> Project Structures …



Interface Engine

```java
package sit.int202.beans;

public interface Engine {
    void turnOn();
    int getCapacity();
    void turnOff();
}
```

Package
sit.int202.beans

```java
public class Car {
    private String chasisNumber ;
    private String brand;
    private Engine engine;
    public Car() {    }
    public Car(String chasisNumber
        , String brand, Engine engine) {
        this.chasisNumber = chasisNumber;
        this.brand = brand;
        this.engine = engine;
    }
    public void start() {
        engine.turnOn();
    }

    @Override
    public String toString() {
        return "Car: "+ chasisNumber + " - " + brand
                + ", " + engine.getCapacity();
    }
}
```

DieselEngine

```
public class DieselEngine implements Engine {
    private int capacity;
    public DieselEngine() {
    }
    public DieselEngine(int capacity) { this.capacity = capacity;  }
    @Override
    public void turnOn() {
        System.out.println("Turn On - Diesel Engine");
    }
    @Override
    public int getCapacity() {  return this.capacity;   }
    @Override
    public void turnOff() {
        System.out.println("Turn Off - Diesel Engine");
    }
}
```

GasolineEngine

```
public class GasolineEngine implements Engine {
    private int capacity;
    public GasolineEngine() {}
    public GasolineEngine(int capacity) {
        this.capacity = capacity;
    }

    @Override
    public void turnOn() {
        System.out.println("Turn On - Gasoline Engine");
    }

    @Override
    public int getCapacity() {  return this.capacity;  }

    @Override
    public void turnOff() {
        System.out.println("Turn Off - Gasoline Engine");
    }
}
```

Car Application without IoC

Package
sit.int202

```
public class Test {
    public static void main(String[] args) {
        tightCoupling();
        // looseCoupling();
    }
    private static void tightCoupling() {
        Engine ge = new GasolineEngine(3000);
        Car carA = new Car("ZB25478-23958D", "Toyota", ge);
        carA.start();
        System.out.println(carA);
    }
}
```

Change Coding to Spring IoC Application

Map Spring beans to POJO classes with **Constructor Injection**

ApplicationConfig.xml

```
<bean id="car" class="sit.int202.beans.Car">
   <constructor-arg name="chasisNumber" value="ZE3197-9485M"/>
   <constructor-arg name="brand" value="Toyota"/>
   <constructor-arg name="engine" ref="2ZZ-GE"/>
</bean>
<bean id="1KD-FTV" class="sit.int202.beans.DieselEngine">
   <constructor-arg value="2982" type="int"/>
</bean>
<bean id="2ZZ-GE" class="sit.int202.beans.GasolineEngine">
   <constructor-arg value="2498" type="int"/>
</bean>
```

Literal Injection

Object Injection

Use the beans from main class

Test::main which *loosely coupled* with Car Details.

```
public class Test {
    public static void main(String[] args) {
        // tightCoupling();
        looseCoupling();
    }

    private static void looseCoupling() {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("ApplicationConfig.xml");
        Car car = (Car) context.getBean("car");
        car.start();
        System.out.println(car);
    }
}
```

Setter Injection

```
<bean id="1KD-FTV" class="sit.int202.beans.DieselEngine">
    <property name="capacity" value="2982"/>
</bean>

<bean id="carX" class="sit.int202.beans.Car">
    <property name="chasisNumber" value="ZE3197-9485M"/>
    <property name="brand" value="Toyota"/>
    <property name="engine" ref="1KD-FTV"/>
</bean>
```

Code based Configuration

```
package sit.int202.config;
@Configuration
public class ApplicationConfig {
    @Bean(name = "car")
    public Car getCar() {
        return new Car("ZM4969JXX", "Toyota-Fortuner", fortunerEngine());
    }
    @Bean(name = "1KD-FTV")
    public Engine fortunerEngine() {
        return new DieselEngine(2499);
    }
}
```

```
private static void usingCodeBasedConfiguration() {
   ApplicationContext ct = new AnnotationConfigApplicationContext(ApplicationConfig.class);
   Car car = (Car) ct.getBean("car");
   car.start();
   System.out.println(car);
}
```