https://www.javatpoint.com/spring-tutorial

SPRING

revision

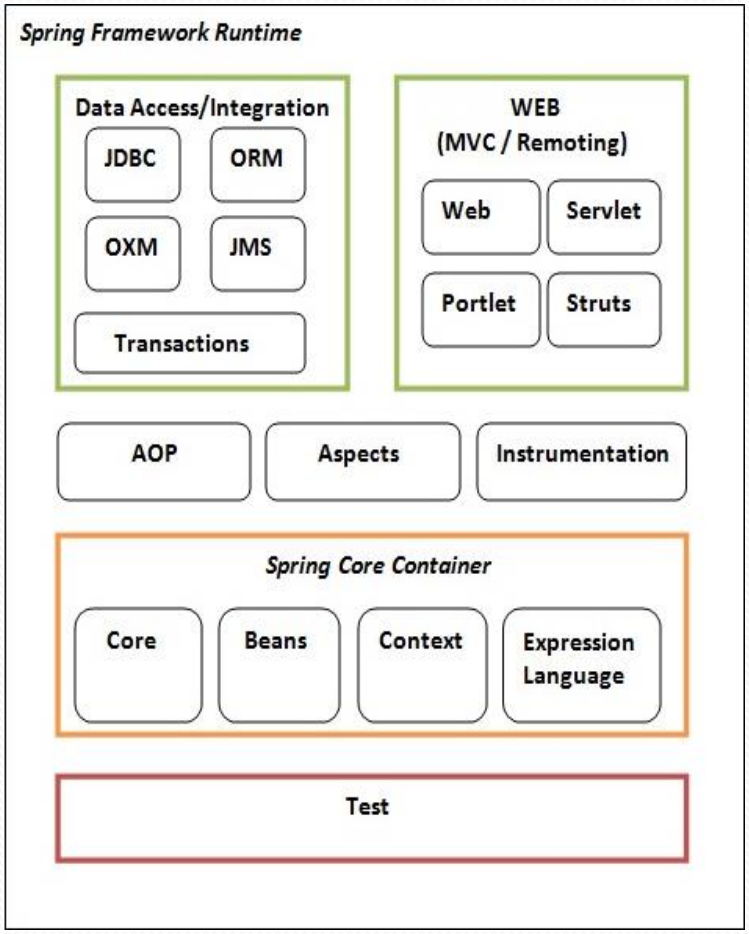
Spring Framework

- Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as:
  - Struts,
  - Hibernate,
  - Tapestry,
  - EJB,
  - JSF, etc.

- A structure where we find solution of the various technical problems.

Spring Modules

- Test
  - This layer provides support of testing with JUnit and TestNG.
- Spring Core Container
  - The Spring Core container contains core, beans, context and expression language (EL) modules
    - Core and Beans
      - These modules provide Inversion of Control and Dependency Injection features.
    - Context
      - It builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured.
    - Expression Language
      - It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.



Spring Modules

- AOP, Aspects and Instrumentation
  - These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.
  - The aspects module provides support to integration with AspectJ.
  - The instrumentation module provides support to class instrumentation and classloader implementations.
- Data Access / Integration
  - This group comprises of JDBC, ORM, OXM, JMS and Transaction modules. These modules basically provide support to interact with the database.
- Web
  - This group comprises of Web, Web-Servlet, Web-Struts and Web-Portlet. These modules provide support to create web application.

Steps to make a very basic Spring Application:

- Load the spring jar files
- Create the class
- Create the xml file to provide the values
- Create the test class
- Run the test class

Step 1: Add the Jars to Classpath

- Core JARs
- All the JARs!

Step 2: Create the Java Class

```
public class Student {
private String name;
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public void displayInfo(){
    System.out.println("Hello: "+name);
}
}
```

Step 3: Create the XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="studentbean" class="springtest.Student">
<property name="name" value="Your Name"></property>
</bean>

</beans>
```

Step 4: Create the Test Class

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class Test {
public static void main(String[] args) {
    Resource resource=new ClassPathResource("applicationContext.xml
");
    BeanFactory factory=new XmlBeanFactory(resource);

    Student student=(Student)factory.getBean("studentbean");
    student.displayInfo();
}
}
```

Inversion of Control (IOC)

- Inversion of control (IoC) is a programming technique in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. In order for the assembler to bind objects to one another, the objects must possess compatible abstractions.

- is a concept in application development
- "don’t call me, I’ll call you"
- one form is Dependency Injection (DI)

Inversion of Control (IOC)

- The Spring container is at the core of the Spring Framework.
- The Spring container uses dependency injection (DI) to manage the components that make up an application.
- The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

Example

```
class Employee{
Address address;
Employee(){
address=new Address();
}
}
```

(Dependency Lookup)
Tightly Coupled

```
class Employee{
Address address;
Employee(Address address){
this.address=address;
}
}
```

(Dependency Injection)
Loosely Coupled

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Dependency Injection Types

- Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

- Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

IOC Container

- The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets information from the XML file and works accordingly. The main tasks performed by IoC container are:

  - to instantiate the application class
  - to configure the object
  - to assemble the dependencies between the objects

IoC Container Types:

- Spring BeanFactory Container
  - This is the simplest container providing basic support for DI. There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the XmlBeanFactory class.

- Spring ApplicationContext Container
  - The ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory. It adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

BeanFactory Container

```
Resource resource=new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
```

ApplicationContext Container

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

Difference between constructor and setter injection

- Partial Dependency – for example, Constructor has 3 parameters, but you have only one value for a variable. You would prefer setter in that case.
- Overriding – Setter injection overrides the constructor 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

- Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and string values. It works with reference only.
- It requires the less code because we don't need to write the code to inject the dependency explicitly.
- You, as a programmer,

4 Type of Autowiring

- byName
- byType
- constructor
- no

byName Autowiring

- In case of byName autowiring mode, bean id and reference name must be same.
- It internally uses setter injection.
- If you change the name of bean, it will not inject the dependency.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName">
</bean>
```

byType Autowiring

- In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.
- It internally uses setter injection.
- In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.
- But, if you have multiple bean of one type, it will not work and throw exception.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byType"></bean>
```

Constructor Autowiring

- In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.
- If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="constructor"></bean>
```

No Autowiring

- In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="no"></bean>
```

Dependency Injection with Factory Methods

- Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

- **factory-method:** represents the factory method that will be invoked to inject the bean.
- **factory-bean:** represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

>>> Recall Factory Pattern

Factory Pattern

```
public class A {
        public static A getA(){//factory method
         return new A();
        }
}
```

Factory Method Types

- A **static factory** method that **returns instance of its own class**. It is used in **singleton design pattern**.

```
<bean id="a" class="com.javatpoint.A" factory-method="getA"></bean>
```

Factory Method Types

- A **static factory** method that **returns instance of another class**. It is used instance is not known and decided at runtime.

```
<bean id="b" class="com.javatpoint.A" factory-method="getB"></bean>
```

Factory Method Types

- A non-static factory method that returns instance of another class. It is used instance is not known and decided at runtime.

```
<bean id="a" class="com.javatpoint.A"></bean>

<bean id="b" class="com.javatpoint.A"
        factory-method="getB" factory-bean="a">
</bean>
```

SPRING AOP

ASPECT ORIENTED PROGRAMMING

AOP

- **Aspect Oriented Programming (AOP)** compliments OOP 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 do we use it?

- It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic.

- **Problem without AOP:** We can call methods (that maintains log and sends notification) from specific 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.

Where do we use AOP?

- AOP is mostly used in following cases:
  - To provide declarative enterprise services such as declarative transaction management.
  - It allows users to implement custom aspects.

AOP Concepts and Terminology

- **Join point**
- **Advice**
- **Pointcut**
- Introduction
- Target Object
- **Aspect**
- Interceptor
- AOP Proxy
- Weaving

Aspect

- It is a class that contains advices, joinpoints etc.

- *Interceptor is an aspect that contains only one advice.*

Join Point

- Join point is any point in your program such as method execution, exception handling, field access etc.
- Spring supports only method execution join point.

Advice

- Advice represents an action taken by an aspect at a particular join point. There are different types of advices:
- **Before Advice:** it executes before a join point.
- **After Returning Advice:** it executes after a joint point completes normally.
- **After Throwing Advice:** it executes if method exits by throwing an exception.
- **After (finally) Advice:** it executes after a join point regardless of join point exit whether normally or exceptional return.
- **Around Advice:** It executes before and after a join point.

Pointcut

- It is an expression language of AOP that matches join points.

AspectJ Annotations

- **@Aspect** declares the class as aspect.
- **@Pointcut** declares the pointcut expression.

AspectJ Annotations

- **@Before** declares the before advice. It is applied before calling the actual method.
- **@After** declares the after advice. It is applied after calling the actual method and before returning result.
- **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
- **@Around** declares the around advice. It is applied before and after calling the actual method.
- **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.

@Pointcut annotation

- Used to define the pointcut. We can refer the pointcut expression by name also.

```
@Pointcut("execution(* Operation.*(..))")
private void doSomething() {}
```

Name: doSomething

Apply on: functions from Operation class

* Represents any type (any return type, any parameters etc)

More examples

```
@Pointcut("execution(public * *(..))")

@Pointcut("execution(public Operation.*(..))")

@Pointcut("execution(public Employee.set*(..))")

@Pointcut("execution(int Operation.*(..))")
```

More Reading

https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html