Lecture 13

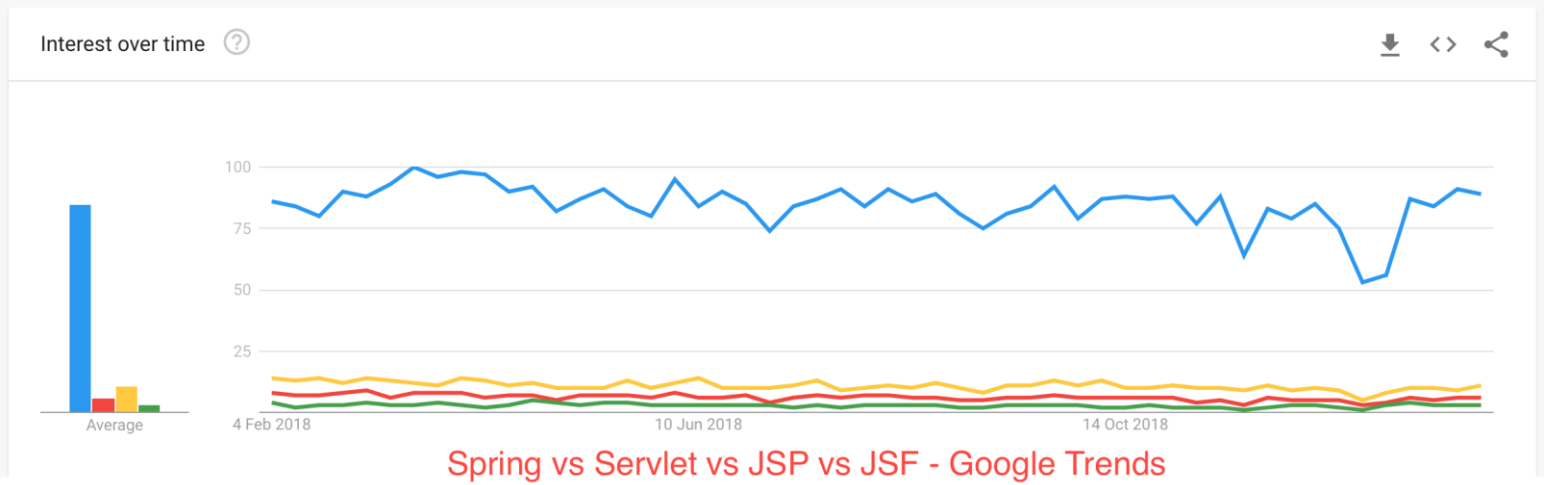
Spring Core Concepts

Contents

- Introduction to Java Spring
- Basic Inversion of Control (IoC)
  - Dependency Injection (DI)
  - Example in Spring Framework
- Basic Aspect-Oriented Programming (AOP)
  - Example in Spring Framework

Introduction to Spring framework

- Spring is the most popular, modern application development framework for enterprise Java (JavaEE)
  - However, Spring only requires Java SE 1.8+
- Spring handles the infrastructure so you can focus on your application



Source: Google Trends

Why Spring framework?

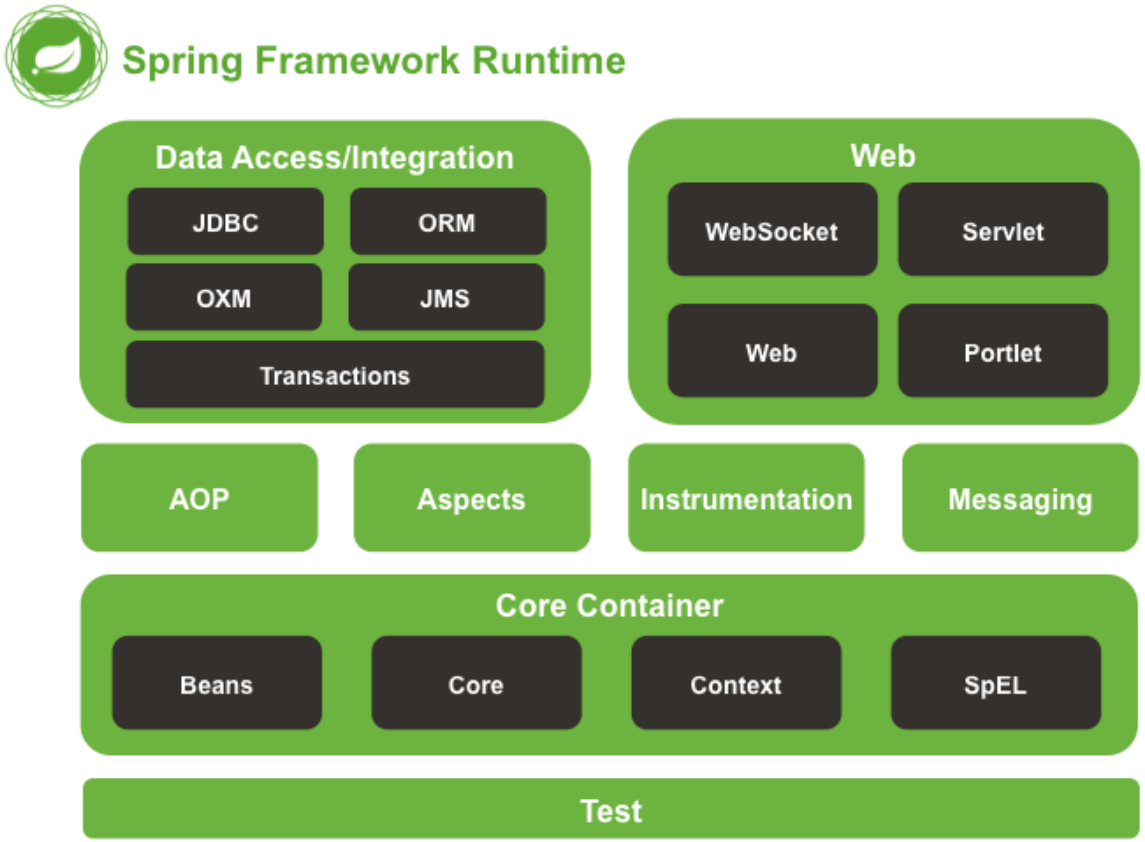
- **POJO**s (Plain Old Java Object), simple to build enterprise-class applications
- **Dependency injection**, promote loose coupling (dependencies)
- **AOP** (Aspect Oriented Programming), allow separation of cross-cutting concerns from the business logic
  - Cross-cutting: something that applies to many parts of your application
  - Logging, declarative transactions, security, caching, etc.

Why Spring framework?

- Minimize the **boilerplate Java code** with helper packages & classes
- Make use **existing technologies**
  - ORM frameworks, logging frameworks, JEE, Quartz, JDK timers...
- Extensions for **web applications**
  - Core features can be used in any application
- Many more features and best practices...

Spring Architecture

- Spring features are organized into about 20 modules.
- These modules are grouped into Core Container, Data Access/Integration, Web, AOP, Instrumentation, and Test.



Setting up Spring project

- Create a Maven project and add the following dependencies in `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.23</version>
    </dependency>
</dependencies>
```

- Reload the Maven project to resolve dependencies if necessary.

Dependency Injection (DI)

- Heart of Spring framework
- An injection is the passing of a dependency (a service) to a dependent object (a client).
- Passing the service to the client, rather than allowing the client to build or find the service, is the fundamental requirement of the pattern.
- "*Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods*" (Wikipedia)

Inversion of Control (IoC)

- In software engineering, inversion of control (IoC) describes a design in which custom written portions of a computer program receive the flow of control from a generic, reusable library.
- The framework calls into the custom, or task-specific code (your code)
- **IoC** is a general concept, where the dependency injection is one concrete example of it.

Benefits of IoC

- Loosely coupled dependencies.
- Better testability for your classes.
- Source code relies on abstraction other than concrete implementations → easier to change underlying implementations.

Simple IoC example

```
lect13.simpleIoC.FirstTry
lect13.simpleIoC.SecondTry
lect13.simpleIoC.ThirdTry
```

Legacy example

- Using a service in the plain old way.

```
public class EmailService {
    public void sendEmail(String message, String receiver) {
        //logic to send email
        System.out.println("Email sent to "
                + receiver + " with Message=" + message);
    }
}

public class MyApplication {
    private EmailService email = new EmailService();

    public void processMessages(String msg, String rec) {
        //do some msg validation, manipulation logic etc
        this.email.sendEmail(msg, rec);
    }
}
```

Dependency Injection example

- Framework-provided codes (package `lect13.framework`)

```
public interface MessageService {
    void sendMessage(String msg, String rec);
}
```

```
public interface Consumer {
    void processMessages(String msg, String rec);
}
```

```
public interface MessageServiceInjector {
    public Consumer getConsumer();
}
```

Dependency Injection example

- Framework-provided codes (package `lect13.framework`)
- Framework needs a way to discover your custom codes.

```
import lect13.di.MyDIApplication;
import lect13.di.SMSMessageService;

public class SMSServiceInjector implements MessageServiceInjector {
    @Override
    public Consumer getConsumer() {
        return new MyDIApplication(new SMSMessageService());
    }
}
```

Dependency Injection example

- Your custom code (package `lect13.di`)
- Your code needs to import classes and interfaces from framework.

```
import lect13.framework.MessageService;

public class SMSMessageService implements MessageService {
    @Override
    public void sendMessage(String msg, String receiver) {
        //logic to send SMS
        System.out.println("SMS sent to "
                + receiver + " with Message=" + msg);
    }
}
```

Dependency Injection example

- Your custom code (package `lect13.di`)

```
import lect13.framework.Consumer;
import lect13.framework.MessageService;

public class MyDIApplication implements Consumer {
    private MessageService service;

    public MyDIApplication(MessageService svc) {
        this.service = svc;
    }

    @Override
    public void processMessages(String msg, String rec) {
        // do some msg validation, manipulation logic etc
        this.service.sendMessage(msg, rec);
    }
}
```

Pros & Cons of Dependency Injection

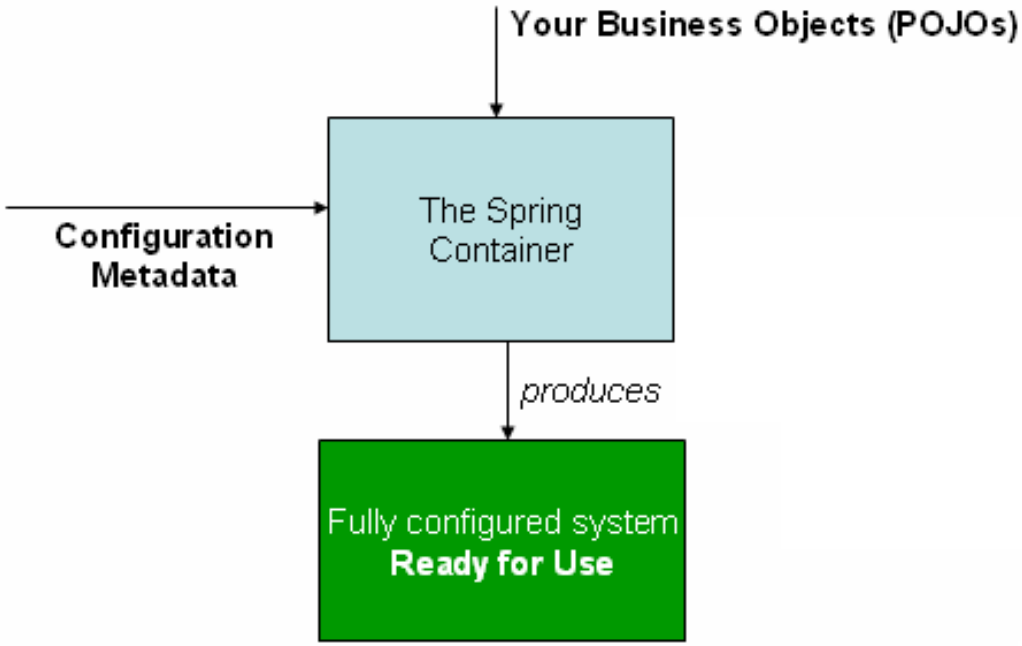
- Benefits
  - Separation of concerns
  - Boilerplate code reduction in application classes
  - Easy to extend application (configurable components)
- Disadvantages
  - Code is difficult to understand
  - Developers become dependent on libraries (if there is a problem, it’s very difficult, often impractical, to fix)
  - Runtime errors cannot be detected at compile time.

Spring IoC Container

- The Spring container (IoC Container) is at the core of the Spring Framework.
- The container will **create** the objects, **wire** them together, **configure** them, and **manage** their complete lifecycle from creation till destruction.

Spring IoC Container

- 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
  - Java code



Spring IoC Example

tutes.ioc.Config

```
@Configuration
public class Config {
    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store myStore() {
        return new Store();
    }
}
```

Spring IoC Example

tutes.ioc.Item

```
public interface Item {
    void sayName();
}
```

Spring IoC Example

tutes.ioc.ItemImpl1

```
public class ItemImpl1 implements Item {
    public void sayName() {
        System.out.println("Item Name");
    }
}
```

Spring IoC Example

tutes.ioc.Store

```
public class Store {
    @Autowired
    private Item item1;

    public Item getItem1() {
        return item1;
    }

    public void greetings() {
        System.out.println("Welcome to the store!");
    }
}
```

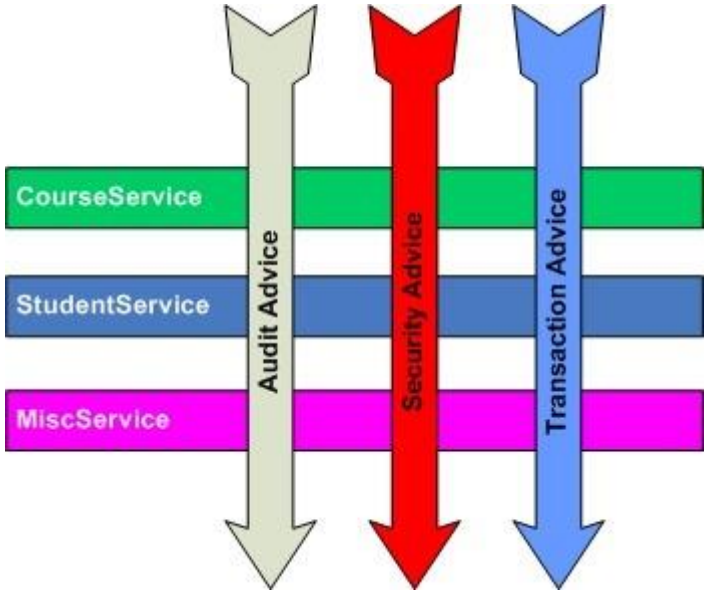
Spring IoC Example

tutes.ioc.Main

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
                new AnnotationConfigApplicationContext(Config.class);
        Store s = context.getBean("myStore", Store.class);
        s.greetings();
        s.getItem1().sayName();
    }
}
```

Aspect Oriented Programming (AOP)

- AOP involves breaking down program logic into distinct parts called **concerns**.
- The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic.
- Examples:
  - Logging
  - Security
  - Transaction
  - Caching



Core concepts of AOP

- **Aspect:** a class that implements application concerns that cut across multiple classes.
- **Join Point:** a specific point in the application such as method execution, exception handling, changing object variable values, etc.
- **Advice:** actions taken for a particular join point. In terms of programming, they are methods that get executed when a certain join point with matching pointcut is reached in the application.
- **Pointcut:** an expression that is matched with a join point to determine whether advice needs to be executed or not.

Spring AOP

- Spring AOP module provides interceptors to intercept an application.
  - E.g. when a method is executed, you can *add extra functionality before or after the method execution*.
- Spring's AOP approach aims to provide a close integration between AOP implementation and Spring IoC (which differs from most frameworks)
- Spring's AOP functionality is normally used in conjunction with the Spring IoC container.
  - Aspects are configured using bean definition syntax.

Spring AOP Advice Types

- **Before advice:** runs before the execution of join point methods.
  - Use `@Before` to mark an advice as Before advice.
- **After (finally) advice:** gets executed after the join point method finishes executing, whether normally or by throwing an exception.
  - Create After advice using `@After` annotation.
- **After Returning advice**: execute only if the join point method executes normally.
  - Use `@AfterReturning` annotation to mark a method as After Returning advice.

Spring AOP Advice Types

- **After Throwing advice:** gets executed only when join point method throws exception.
  - Can be used to rollback transaction.
  - Use `@AfterThrowing` annotation for this type of advice.
- **Around advice**: this advice surrounds the join point method and we can also choose whether to execute the join point method or not.
  - The most important and powerful advice.
  - We can write advice code that gets executed before and after the the join point.
  - Around advice invokes the join point method and return values if the method is returning something.
  - Use `@Around` annotation to create Around advice methods.

Setting up AOP for Spring project

- In `pom.xml`, add the AOP library into the list of dependencies:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.23</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.9.1</version>
    </dependency>
</dependencies>
```

(*) Reload the Maven project to resolve dependencies if necessary.

Spring AOP Example

org.example.Config

```
@EnableAspectJAutoProxy
@ComponentScan("org.example")
public class Config {

}
```

Spring AOP Example

org.example.EmailMessageService

```
@Component
public class EmailMessageService {
    public void sendMessage(String msg, String receiver) {
        System.out.println("Message '" + msg
                + "' has been sent to '" + receiver + "'.");
    }
}
```

Spring AOP Example

org.example.ConsumerApp

```
@Component
public class ConsumerApp {
    @Autowired
    private EmailMessageService svc;

    public void processMessage(String msg, String rec) {
        //some magic like validation, logging etc
        svc.sendMessage(msg, rec);
    }
}
```

Spring AOP Example

org.example.Bootstrap

```
public class Bootstrap {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context
                = new AnnotationConfigApplicationContext(
                Config.class);
        ConsumerApp app
                = context.getBean(ConsumerApp.class);
        app.processMessage("Hello", "quandd@hanu.edu.vn");
    }
}
```

Spring AOP Example

org.example.Logger

```
@Component
@Aspect
public class Logger {
    @Pointcut("execution(* sendMessage(..))")
    public void someMethod() {
    }
    @Before("someMethod()")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Logging 1...");
    }
    @After("execution(* sendMessage(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("Logging 2...");
    }
}
```