COMP S380F Lecture 10: Dependency Injection (DI), Aspect-Oriented Programming (AOP)

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

- *Decoupling* an interface from its implementation
- Dependency Injection (DI)
  - Construction injection
  - Setter injection
- Spring IoC (Inversion of Control) Container
- Spring Application Context
- Spring Bean
  - Inner beans
- Wiring Collections: List, Set, Map
- Autowiring
- Aspect-Oriented Programming (AOP)

Enterprise JavaBean (EJB)

- Class libraries only support the sharing of OO code.
- Java EE has EJBs, which support the sharing of code and data.
  - Different OO applications can share states of objects running on the server.
- Java EE supports many services through EJBs including transactions processing, Java persistence API (JPA), concurrency control, naming and directory services (JNDI), security and web services.
- Programmer can invoke the services by calling the EJB’s complex API.

Plain Old Java Object (POJO)

- The use of POJO allows Spring programmers to write code without too much changes to their coding style (EJBs do not allow their programmers to use them in a simple way).
- Spring and Hibernate make good use of POJOs.
- Under the influence from Hibernate and Spring, EJB 3 has improved over EJB 2 for being simpler.

POJO without an Interface

- The following class tries to perform a binary operation.
- The current binary operation is addition.
- If we want to change the binary operation to multiplication, we will need to change the bodies of two methods to change the operation.
- This code is not very maintainable.

```
public class Calculate {
   private long operate (long op1, long op2) {
      return op1 + op2;
   }

   private String getOpsName() {
      return " plus ";
   }
}
```

POJO with an Interface and Implementations

- To make the previous code more maintainable, we can define an **interface** for the binary operation.

```
public interface Operation {
   long operate (long op1, long op2);
   String getOpsName();
}
```

- We can have different **implementations** for the Operation interface, e.g., OpAdd and OpMultiply below.

```
public class OpAdd implements Operation {
    public OpAdd() {}
    @Override
    public String getOpsName() {
        return " plus ";
    }
    @Override
    public long operate(long op1, long op2) {
        return op1 + op2;
    }
}
```

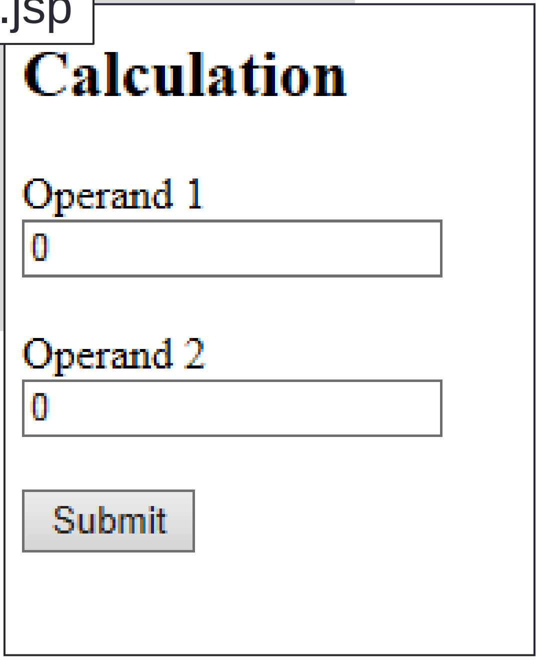
```
public class OpMultiply implements Operation {
    public OpMultiply() {}
    @Override
    public String getOpsName() {
        return " times ";
    }
    @Override
    public long operate(long op1, long op2) {
        return op1 * op2;
    }
}
```

Web application example

- Webapp example: **Lecture10**
- URL: <base URL>/calEx1

```
@GetMapping("/calEx1")
public ModelAndView showCalForm1() {
    return new ModelAndView("cal", "calForm", new CalForm());
}

public static class CalForm {
    private long op1;
    private long op2;

    // getters and setters for op1, op2
}
```

cal.jsp



Calculation

Operand 1

0

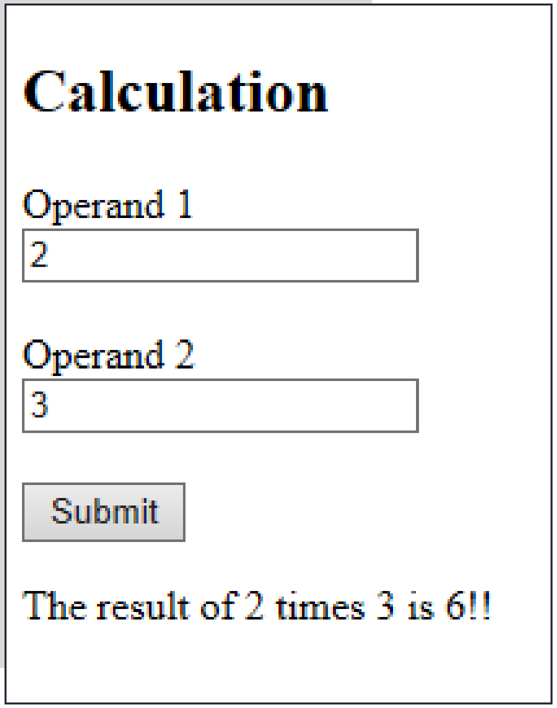
Operand 2

0

Caller choosing implementation

- Example URL: <base URL>/calEx1
- Just change one bold statement, we can switch to perform another operation, e.g., **Operation ops = new OpAdd();**

```
@PostMapping("/calEx1")
public ModelAndView cal1(CalForm calForm) {
    Operation ops = new OpMultiply();
    long op1 = calForm.getOp1();
    long op2 = calForm.getOp2();

    String result = "The result of " + op1
                + ops.getOpsName() + op2 + " is "
                + ops.operate(op1, op2) + "!!";

    ModelAndView mav = new ModelAndView("cal");
    mav.addObject("calForm", calForm);
    mav.addObject("result", result);
    return mav;
}
```



Calculation

Operand 1

2

Operand 2

3

The result of 2 times 3 is 6!!

Interface requires code change

- The use of interface can make a program more versatile and maintainable by choosing the desired implementation.
- Yet the caller must **change the code to switch implementations** in this interface solution.
  - ➢ Updating Java code requires re-compilation of the web application.
- Spring allows us to select implementation by **modifying the XML configuration file** for the Spring application context.
  - ➢ Updating XML code does not require re-compilation of the web application (though re-deployment is needed).

Adding Spring beans to dispatcher-servlet.xml

- We can define Spring beans in the Spring application context (which is configured in dispatcher-servlet.xml).

```
<bean id="multiply" class="hkmu.examples.OpMultiply" />
<bean id="add" class="hkmu.examples.OpAdd" />

<bean id="opsbean" class="hkmu.examples.CalculateSpring">
    <property name="ops" ref="add" />
</bean>
```

```
public class CalculateSpring {
    private Operation ops;

    public void setOps(Operation ops) {
        this.ops = ops;
    }

    public String getOpsName() {
        return ops.getOpsName();
    }

    public long operate(long op1, long op2) {
        return ops.operate(op1, op2);
    }
}
```

- For each property, Spring will call the suitable setter method using the value of the attribute “ref” as an object reference.
  - ➢ setOps for property name ops
- We can switch between OpAdd and OpMultiply by setting this “ref” attribute to the bean “add” or the bean “multiply”

Getting Spring bean in Controller method

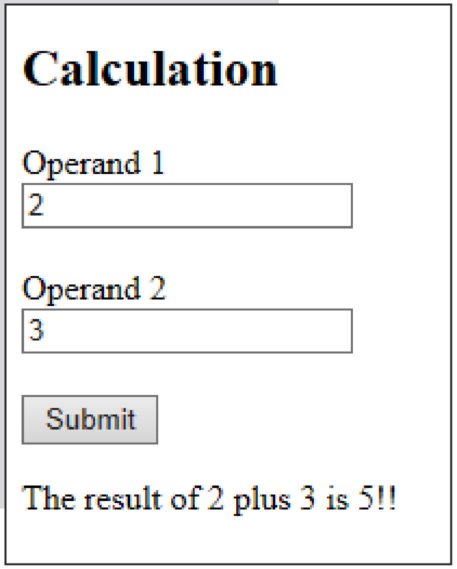
- Example URL: <base URL>/calEx2
- In a controller method, we can get the Spring bean “opsbean” using the **ApplicationContext** object and its **getBean()** function, as follows:

```
@Autowired
ApplicationContext context;
```

OR

```
ApplicationContext context;

@Autowired
public ExampleController(ApplicationContext context) {
   this.context = context;
   System.out.println("Application context "
                     +context.toString() +"loaded."); }
```

```
@PostMapping("/calEx2")
public ModelAndView cal2(CalForm calForm) {
    CalculateSpring cs = (CalculateSpring) context.getBean("opsbean");
    long op1 = calForm.getOp1();
    long op2 = calForm.getOp2();

    String result = "The result of " + op1
                + cs.getOpsName() + op2 + " is "
                + cs.operate(op1, op2) + "!!";
    ModelAndView mav = new ModelAndView("cal");
    mav.addObject("calForm", calForm);
    mav.addObject("result", result);
    return mav;
}
```



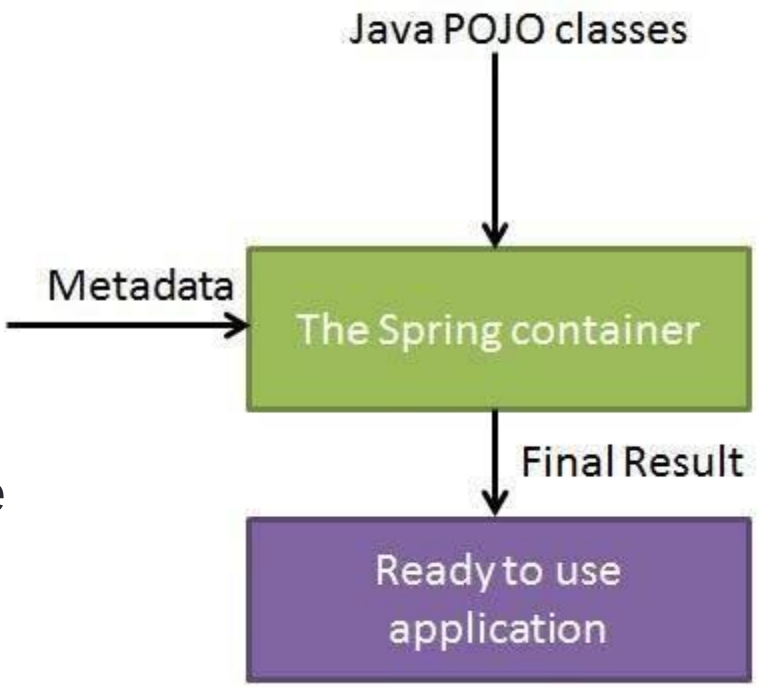
The result of 2 plus 3 is 5!!

Dependency Injection (DI)

- The technique we’ve just shown is called Dependency Injection (DI).
- The dependency is no longer hard-coded in the caller but *injected* externally through a configuration file.
- DI is a form of **Inversion of Control (IoC).**
  - ➢ IoC is a design principle where it is the generic framework that calls into the custom code.
  - ➢ This inverts control as compared to traditional programming, where the custom code calls into reusable libraries to take care of generic task.
  - ➢ “Don’t call us; we’ll call you”.

Spring IoC Container

- The **Spring 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.
- The Spring IoC container uses **dependency injection (DI)** to manage the components that make up an application.
  - ➢ These objects are **Spring Beans**.
- The container gets its instructions on what Spring beans to instantiate, configure, and assemble by reading the provided configuration metadata (XML, Java annotations, or Java code).



Spring Bean

- A Spring bean is instantiated, assembled, and otherwise managed by a Spring IoC container.
- These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML **`<bean />`** definitions which you have already seen previously.

dispatcher-servlet.xml

```
<bean id="multiply" class="hkmu.examples.OpMultiply" />
<bean id="add" class="hkmu.examples.OpAdd" />

<bean id="opsbean" class="hkmu.examples.CalculateSpring">
    <property name="ops" ref="add" />
</bean>
```

More on Spring IoC Containers

- The association made between Spring beans in the XML file are called **wiring**.
- Spring application objects are created, wired, configured inside Spring containers.
- For standalone application, the following Spring application contexts are examples of Spring IoC Containers.

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

ApplicationContext context =
        new AnnotationConfigApplicationContext(BeanConfiguration.class);
```

- For web application, we can get the Spring application context in the controller class:

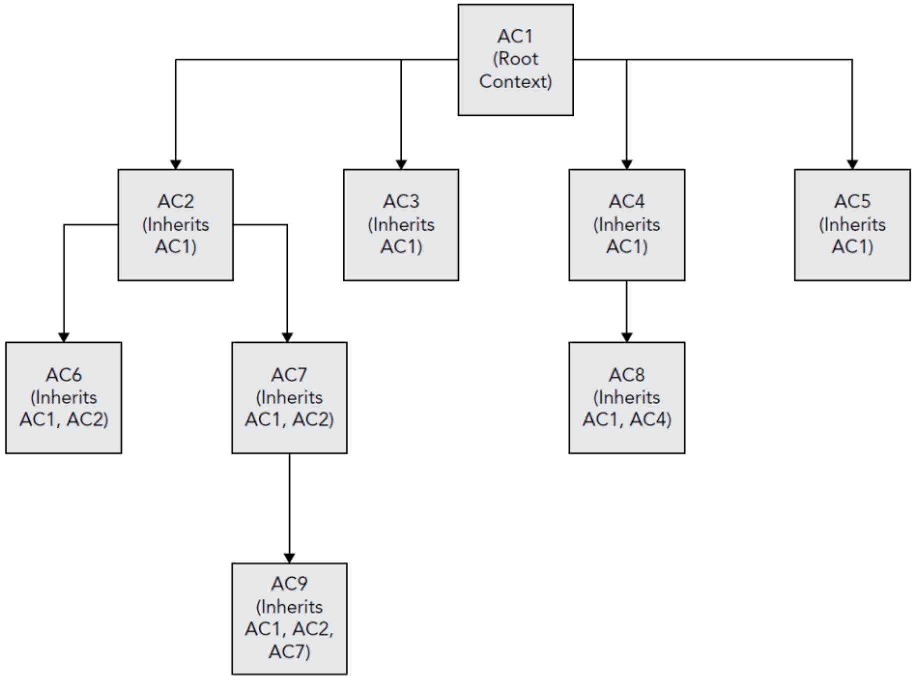
```
@Autowired
ApplicationContext context;
```

Spring Application Context

- Spring IoC container comes in the form of one or more application contexts, represented by the org.springframework.context.ApplicationContext interface.
- A Spring application always has at least one application context, but it can also have a **hierarchy of multiple application contexts**.
  - ➢ E.g., The user and administrative sections of a web app have no access to each other, but they share some application components.
- Spring handles web requests using a **dispatcher servlet**, which delegates incoming requests to the appropriate controllers and translates request and response entities as needed.
- Your web application can have as many instances of the DispatcherServlet class as you want, e.g., for separating your web user interface from your web services.
- **Each DispatcherServlet instance gets its own application context**, which has references to the web application's ServletContext and its own ServletConfig.

Spring Application Context (cont')

- Because none of these DispatcherServlets can access the application context for any other DispatcherServlets, it is often desirable to share certain beans (such as business objects or data access objects) in a common **root application context**.
- This application context, global to the entire web application, is the parent of all the DispatcherServlets' application contexts and is created using the ContextLoaderListener.
- As a general rule, you should always have one root application context from which all other application contexts inherit in one way or another.



More details can be found in:
http://javarticles.com/2016/02/spring-contexthierarchy-annotation-example.html

Two types of Injection: Construction Injection

- Construction injection sets up a bean by its constructor.

```java
public class CalPowerSpring {
    private int power;
    private Operation ops;

    public CalPowerSpring(int power, Operation ops) {
        this.power = power;
        this.ops = ops;
    }

    public String getOpsName() {
        return ops.getOpsName();
    }

    public long operate(long op1, long op2) {
        return ops.operate((long) Math.pow(op1, power), (long) Math.pow(op2, power));
    }

    public String getPower() {
        return String.valueOf(power);
    }
}
```

```xml
<bean id="powerOpsBean"
          class="hkmu.examples.CalPowerSpring">
    <constructor-arg value="5" />
    <constructor-arg ref="add" />
</bean>
```

Construction Injection (cont')

- Construction injection sets up a bean by its constructor.

```
public class CalPowerSpring {
   …
   public CalPowerSpring(int power, Operation ops) {
        this.power = power;
        this.ops = ops;
    }
    …
}
```

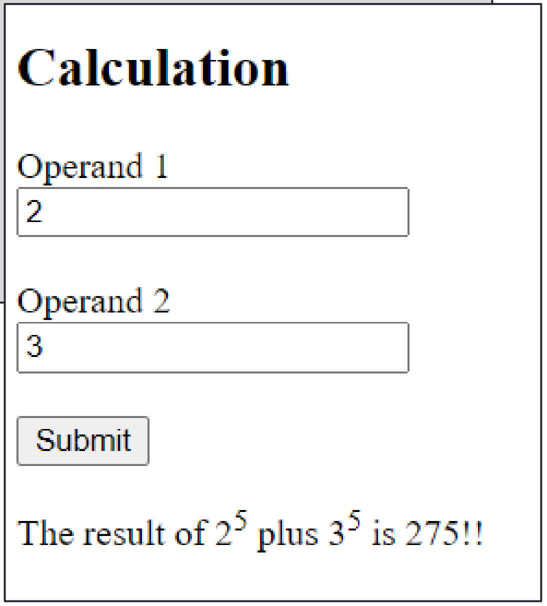
```
<bean id="powerOpsBean"
          class="hkmu.examples.CalPowerSpring">
    <constructor-arg value="5" />
    <constructor-arg ref="add" />
</bean>
```

- We use the **<constructor-arg>** tags to supply arguments to the constructor of CalPowerSpring (**in the same argument order**).
  - ➢ Use the **ref** attribute for passing a reference to an object.
  - ➢ Use the **value** attribute for passing a value directly.

Construction Injection (cont')

- Example URL: <base URL>/calEx3

```
@PostMapping("/calEx3")
public ModelAndView cal3(CalForm calForm) {
    CalPowerSpring cs = (CalPowerSpring) context.getBean("powerOpsBean");
    long op1 = calForm.getOp1();
    long op2 = calForm.getOp2();

    String result = "The result of " + op1 + "<sup>" + cs.getPower() + "</sup>"
                + cs.getOpsName() + op2 + "<sup>" + cs.getPower() + "</sup> is "
                + cs.operate(op1, op2) + "!!";
    ModelAndView mav = new ModelAndView("cal");
    mav.addObject("calForm", calForm);
    mav.addObject("result", result);
    return mav;
}
```



Calculation

Operand 1

2

Operand 2

3

Submit

The result of 2^5 plus 3^5 is 275!!

Two types of Injection: Setter Injection

- Setter injection sets up a bean by calling its setter methods.

```
<bean id="multiply" class="hkmu.examples.OpMultiply" />
<bean id="add" class="hkmu.examples.OpAdd" />

<bean id="opsbean" class="hkmu.examples.CalculateSpring">
    <property name="ops" ref="add" />
</bean>
```

```
public class CalculateSpring {
    private Operation ops;

    public void setOps(Operation ops) {
        this.ops = ops;
    }

    public String getOpsName() {
        return ops.getOpsName();
    }

    public long operate(long op1, long op2) {
        return ops.operate(op1, op2);
    }
}
```

- We use the **<property>** tags:
  - ➢ Use the **ref** attribute for passing a reference to an object.
  - ➢ Use the **value** attribute for passing a value directly.

Example URL:
<base URL>/calEx2

Inner Beans

- In Java, you have inner classes.
- In Spring, you also have inner beans.
- The following inner bean of OpAdd class is used in a setter method of “ops”.

```
<bean id="opsWithInnerBean" class="hkmu.examples.CalculateSpring">
    <property name="ops">
        <bean class="hkmu.examples.OpAdd" />
    </property>
</bean>
```

- Example URL: <base URL>/calEx4

Wiring Collections

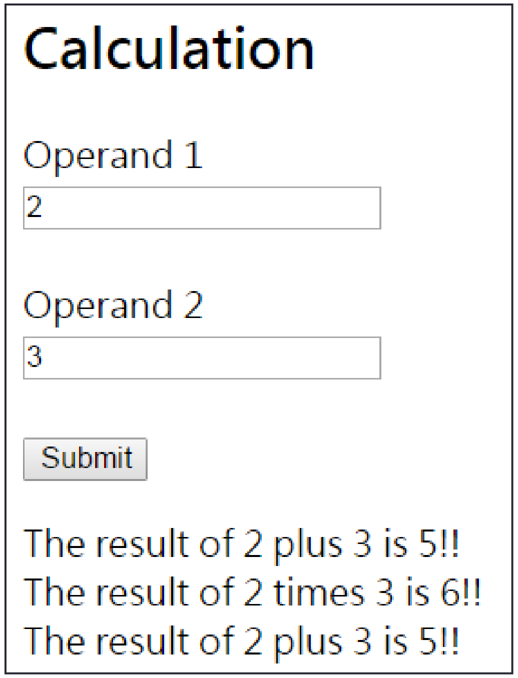
- In previous examples, we switch between individual objects.
- In the following two examples, we switch between different collections of objects.

```
public class CollectionSpring {
    private Collection<Operation> ops;

    public String operateResult(long op1, long op2) {
        String result = "";
        for (Operation op : ops)
            result += "The result of " + op1
                        + op.getOpsName() + op2 + " is "
                        + op.operate(op1, op2) + "!!</br>";
        return result;
    }

    public void setOps(Collection<Operation> ops) {
        this.ops = ops;
    }
}
```

Wiring Collections: List

- A <list> holds possibly duplicated values.

```
<bean id="listBean"
        class="hkmu.examples.CollectionSpring">
    <property name="ops">
        <list>
            <ref bean="add" />
            <ref bean="multiply" />
            <ref bean="add" />
        </list>
    </property>
</bean>
```



Calculation

Operand 1
2

Operand 2
3

Submit

The result of 2 plus 3 is 5!!
The result of 2 times 3 is 6!!
The result of 2 plus 3 is 5!!

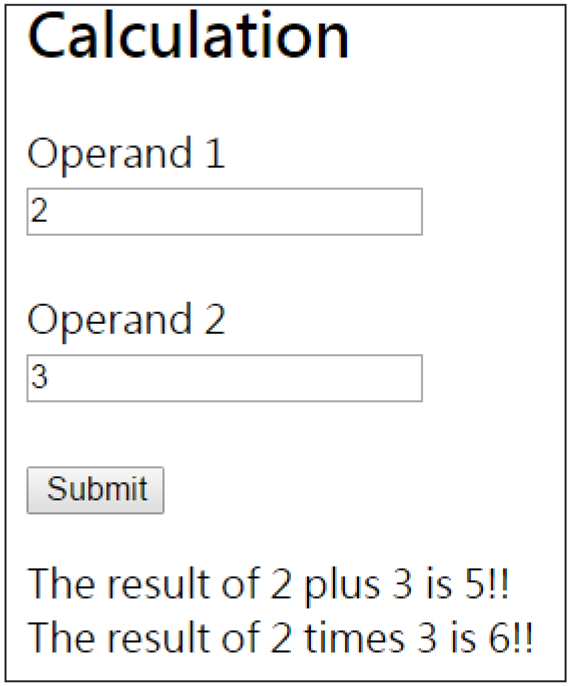
```
@PostMapping("/list")
public ModelAndView calList(CalForm calForm) {
    CollectionSpring cs = (CollectionSpring) context.getBean("listBean");
    long op1 = calForm.getOp1();
    long op2 = calForm.getOp2();

    ModelAndView mav = new ModelAndView("cal");
    mav.addObject("calForm", calForm);
    mav.addObject("result", cs.operateResult(op1, op2));
    return mav;
}
```

Example URL: <base URL>/list

Wiring Collections: Set

- Use <set> to eliminate duplicated values automatically.

```
<bean id="setBean"
        class="hkmu.examples.CollectionSpring">
    <property name="ops">
        <set>
            <ref bean="add" />
            <ref bean="multiply" />
            <ref bean="add" />
        </set>
    </property>
</bean>
```


Calculation

Operand 1

2

Operand 2

3

The result of 2 plus 3 is 5!!
The result of 2 times 3 is 6!!

```
@PostMapping("/set")
public ModelAndView calList(CalForm calForm) {
    CollectionSpring cs = (CollectionSpring) context.getBean("setBean");
    long op1 = calForm.getOp1();
    long op2 = calForm.getOp2();

    ModelAndView mav = new ModelAndView("cal");
    mav.addObject("calForm", calForm);
    mav.addObject("result", cs.operateResult(op1, op2));
    return mav;
}
```

Example URL: <base URL>/set

Wiring Collections: Map

- If ops is a **Map<String, Operation>**, we can use <map>, as follows:

```
<bean id="mapBean" class="hkmu.examples.CollectionSpring">
    <property name="ops">
        <map>
            <entry key="ADD" value-ref="add" />
            <entry key="MULTIPLY" value-ref="multiply" />
        </map>
    </property>
</bean>
```

Property-Value Pairs

- Property-value pairs are often called maps, dictionaries and associations.

```
public class JMSource {
   private Properties sourceProps = null;
   public void setSourceProps(Properties sourceProps) {
      this.sourceProps = sourceProps;
   }
}
```

```
<bean name="jmsSource" class="some.package.JMSSource">
   <property name="sourceProps">
      <props>
         <prop key="numberOfThreads">10</prop>
         <prop key="reuseThreads">true</prop>
         <prop key="collectStatsEvery">hour</prop>
      <props>
   </property>
</bean>
```

Autowiring

```
<bean id="ops" class="hkmu.examples.OpMultiply" />
```

- Given the above bean, the following two declarations are the same.

```
<bean id="opsbean" class="hkmu.examples.CalculateSpring">
    <property name="ops" ref="ops" />
</bean>
```

```
<bean id="opsbean" class="hkmu.examples.CalculateSpring" autowire="byName" />
```

- The property name “ops” will be wired automatically to bean id “ops” for the matching name.
- Other possible values for the autowire attribute are byType, constructor and autodetect.
- As we have seen previously, we can also use the annotation **`@Autowired`** when declaring the property in the Java class for autowiring.

Bean Scoping

- When declaring a bean in Spring, we can choose a scope for the bean to support a specific kind of sharing. The default is singleton.

```
<bean id=“multiply"
        class="hkmu.examples.OpMultiply"
        scope="prototype" />
```

Scope	Meaning
singleton	One instance of the bean per Spring container
prototype	A new instance for every use of the bean
request	One instance per HTTP request
session	One instance per HTTP session
application	One instance for the application
websocket	One instance for a particular WebSocket session

Ref: https://docs.spring.io/spring-framework/docs/5.3.x/reference/html/core.html#beans-factory-scopes

Creating Beans from Factory Method

```
public class Stage {
   private Stage() {}

   private static class StageSingletonHolder {
      static Stage instance = new Stage();
   }

   public static Stage getInstance() {
      return StageSingletonHolder.instance;
   }
}
```

- Why use the *getInstance()* method instead of *new*?
- Delaying creation of an object or the evaluation of an expression is called **lazy evaluation**, which is often used in functional languages.
- It supports singleton pattern when one object is enough.

```
<bean id="theStage"
   class="somepackage.Stage"
   factory-method="getInstance" />
```

Initializing and Cleaning up a Bean

```
public void tuneInstrument() {
   instrument.tune();
}

public void cleanInstrument() {
   instrument.clean();
}
```

- Assuming that the above methods are defined in the Instrumentalist class.
- You can define the bean’s initialization and cleanup, as follows.

```
<bean id="kenny"
      class="somepackage.Instrumentalist"
      init-method="tuneInstrument"
      destroy-method="cleanInstrument">
  <property name="song" value="Jingle Bells" />
  <property name="instrument" ref="saxophone" />
</bean>
```

Aspect-Oriented Programming (AOP)

- The users use an application to perform the tasks they want, e.g., transfer money from one account to another. The users' tasks are the application's **main functions**.
- **Cross-cutting concerns** are not the user tasks they want to perform but are important nevertheless, e.g., logging, security and transaction management.
- **Aspect-Oriented Programming (AOP)** entails breaking programming logic into distinct parts, i.e., main functions and cross-cutting concerns.

More terminology:

- **Pointcuts** are the places in the main user tasks ***where*** cross-cutting concerns should be addressed.
- **Advices** are the code ***how*** to handle the cross-cutting concerns.
  - ➢ E.g., advices are the code to do logging, security, etc.
- An **aspect** consists of all the pointcuts and advices relating to a cross-cutting concern.

Without AOP vs. With AOP

Without AOP:

- The code for the application’s main functions will be intermixed with the code for the cross-cutting concerns.
- The resulting code will be hard to maintain (because when you need to change the main functions, you will have to sieve through a large amount of unrelated cross-cutting concern code and vice versa).
- Also, it is harder to delegate the main function code and cross-cutting concern code to different programmers due to the undesirable intermixing.

With AOP:

- The codes of main function and cross-cutting concern are placed in separate files.
- It is the responsibility of the AOP framework to weave them together.

Example: Main function

- Consider the following money transfer operation, whose job is just to withdraw money from an account and deposit the same amount to another account.

```
void transfer(Account fromAcc, Account toAcc, int amount)
 throws Exception {
   if (! checkUserPermission(user)){
      throw new UnauthorizedUserException();
   }

   if (fromAcc.getBalance() < amount){
      throw new InsufficientFundsException();
   }

   fromAcc.withdraw(amount);
   toAcc.deposit(amount);
}
```

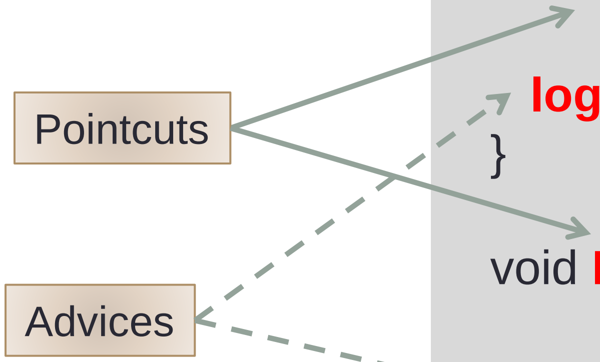
Example: Main function Intermixed with Logging

- After adding the required logging code, the original transfer code is cluttered by the logging code.

```
void transfer(Account fromAcc, Account toAcc, int amount)
 throws Exception {
   logger.info("transferring money...");
   if (! checkUserPermission(user)){
      logger.info("User has no permission.");
      throw new UnauthorizedUserException();
   }
   if (fromAcc.getBalance() < amount){
      logger.info("Insufficient Funds, sorry");
      throw new InsufficientFundsException();
   }
   fromAcc.withdraw(amount);
   toAcc.deposit(amount);
   logger.info("Successful transaction.");
}
```

Example: Extract Logging as an Aspect

- With the following aspect (in pseudo-code), we don’t need to litter the core code with **the cross-cutting concern logging** as on the last slide.

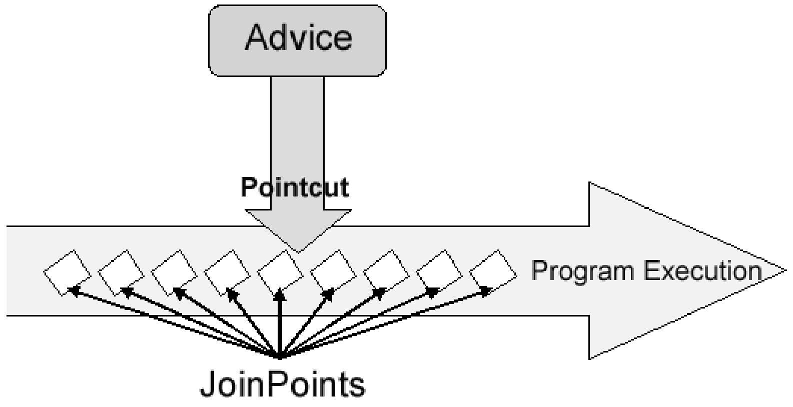


```
aspect Logger {
  void Bank.transfer(Account fromAcc, Account toAcc,
            int amount, User user, Logger logger) {
    logger.info("transferring money...");
  }

  void Bank.getMoneyBack(User user, int transactionId,
            Logger logger) {
    logger.info("User requested money back");
  }

  // other crosscutting code...
}
```

- A **pointcut** indicates where and when the cross-cutting concern applies.
- An **advice** describes what to do.

Steps of using AOP

- Write the **main functions** (user tasks).
- Write an **aspect** (e.g., security and logging) which has
  - **pointcut**: where and when the cross-cutting concern applies.
  - **advice**: the actions of the cross-cutting concern.
- The main functions, pointcuts and advices are separated clearly resulting in cleaner code.
- The framework, e.g., Spring, will **weave** the separated code together as an executable.
- Spring supports two approaches for custom aspects.
  - @AspectJ annotation
  - Schema-based



Defining Logging Aspect in Spring

- We use the @AspectJ annotation approach.

1. In pom.xml, add Maven dependencies for AspectJ:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>${aspectj.version}</version>
</dependency>
```

2. In dispatcher-servlet.xml, add the Spring namespace “aop” and the following configuration tags:

```
<beans  …   xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation=" …  http://www.springframework.org/schema/aop
                              http://www.springframework.org/schema/aop/spring-aop.xsd">
   <aop:aspectj-autoproxy />
   <bean id="logaspect" class="hkmu.examples.aspect.LoggingAspect" />
</beans>
```

Defining Logging Aspect in Spring (cont')

- We use the @AspectJ annotation approach.

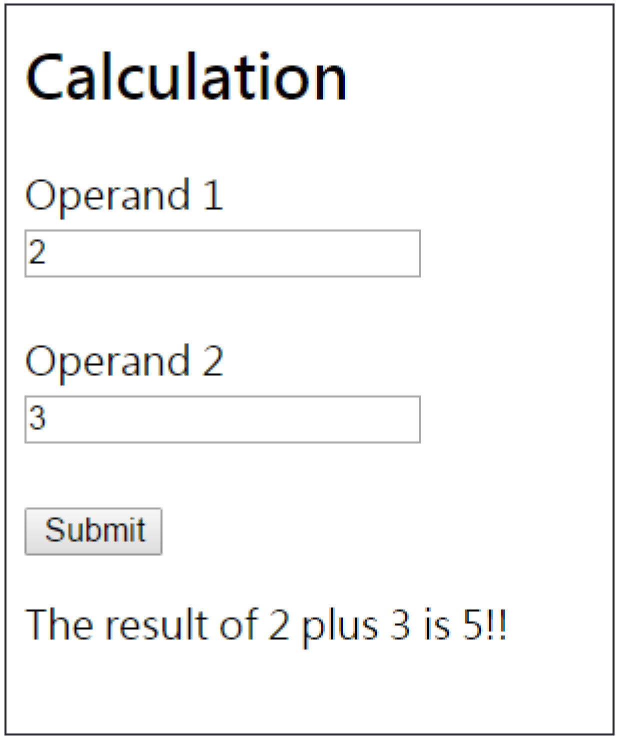
3. In the package "hkmu.examples.aspect", define the aspect class LoggingAspect:

```
package hkmu.examples.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class LoggingAspect {
    // intercept any method execution in the class
    @Before("execution(* hkmu.examples.Operation.*(..))")
    public void logMethodExecution(JoinPoint jp) {
        System.out.println("AOP logging: " + jp.toShortString());
    }
}
```

Note that all other main classes are **unaffected** by AOP.

Defining Logging Aspect in Spring (cont’)

- Use the browser to access <base URL>/calEx2
- Perform the operation in the right.



2

3

The result of 2 plus 3 is 5!!

- The following is displayed in the output of Apache Tomcat:

```
AOP logging: execution(Operation.getOpsName())
AOP logging: execution(Operation.operate(..))
```