Spring - AOP

Aspect-Oriented Programming (AOP)

- *AOP* complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure
- The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*
- Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (*crosscutting* concerns)

Crosscutting Concerns

- These cross-cutting concerns are pieces of logic that have to be applied at many places but actually don't have anything to do with the business logic
- Eg: logging, performance tracking, transaction management

Advantages of AOP

- The logic for each concern is now in one place, as opposed to being scattered all over the code base.
- classes are cleaner since they only contain code for their primary concern (or core functionality) and secondary concerns have been moved to aspects.

Example – OOP Style

```
void setX(...) {
        ...
        ...
        Display.update();
}

void setY(...) {
        ...
        ...
        Display.update();
}

void set...(...) {
        ...
        ...
        Display.update();
}
```

Example – AOP Style

Aspect:

```
after() : set() {
        Display.update();
}
```

Pointcut:

```
pointcut set() : execution(* set*(*) ) &&
        this(MyGraphicsClass) &&
        within(com.company.*);
```

AOP concepts

- ***Aspect*:** a class that contains advices, joinpoints etc.,
- ***Join point*:** a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
- **Advice:** action taken by an aspect at a particular join point.
- **Pointcut:** a expression that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).
- **Interceptor:** An aspect that has only one advice

AOP Concepts

- **Introduction:** declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object.
- **Target object:** object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.
- **AOP proxy:** an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- **Weaving:** linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice

- ***Before advice*:** Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- ***After returning advice*:** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
- ***After throwing advice*:** Advice to be executed if a method exits by throwing an exception.
- ***After (finally) advice*:** Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- ***Around advice*:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception

Aspect - Example

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class EgAspect {
}

<bean id="myAspect“ class="org.xyz.EgAspect">
</bean>
```

Pointcut - Example

```
// the pointcut expression
@Pointcut("execution(* transfer(..))")
// the pointcut signature
private void anyOldTransfer() {
}
```

Pointcut Designators

- *execution* - for matching method execution join points
- *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- *this* - limits matching to join points (the execution of methods) where the bean reference (Spring AOP proxy) is an instance of the given type
- *target* - limits matching to join points (the execution of methods) where the target object (application object being proxied) is an instance of the given type
- *args* - limits matching to join points (the execution of methods) where the arguments are instances of the given types

Declaring Before advice

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

@Aspect
public class BeforeExample {
        @Before("com.xyz.myapp.DAO.dataAccessOpr()")
        public void doAccessCheck() {
                // ...
        }
}
```

After advice

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {
        @AfterReturning("com.xyz.DAO.dataAccessOpr()")
        public void doAccessCheck() {
                // ...
        }
}
```

After advice with return value

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

        @AfterReturning(
                pointcut="com.xyz.DAO.dataAccessOperation()",
                returning="retVal")
        public void doAccessCheck(Object retVal) {
                // ...
        }
}
```

Combining pointcut expressions

```
@Pointcut("execution(public * (..))")
private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```