JAVA FULL STACK DEVELOPMENT PROGRAM

Session 22: Spring AOP & logging

OUTLINE

- Spring AOP
  - AOP Concept
  - Proxy
  - AspectJ
  - Controller Advice
- Logging

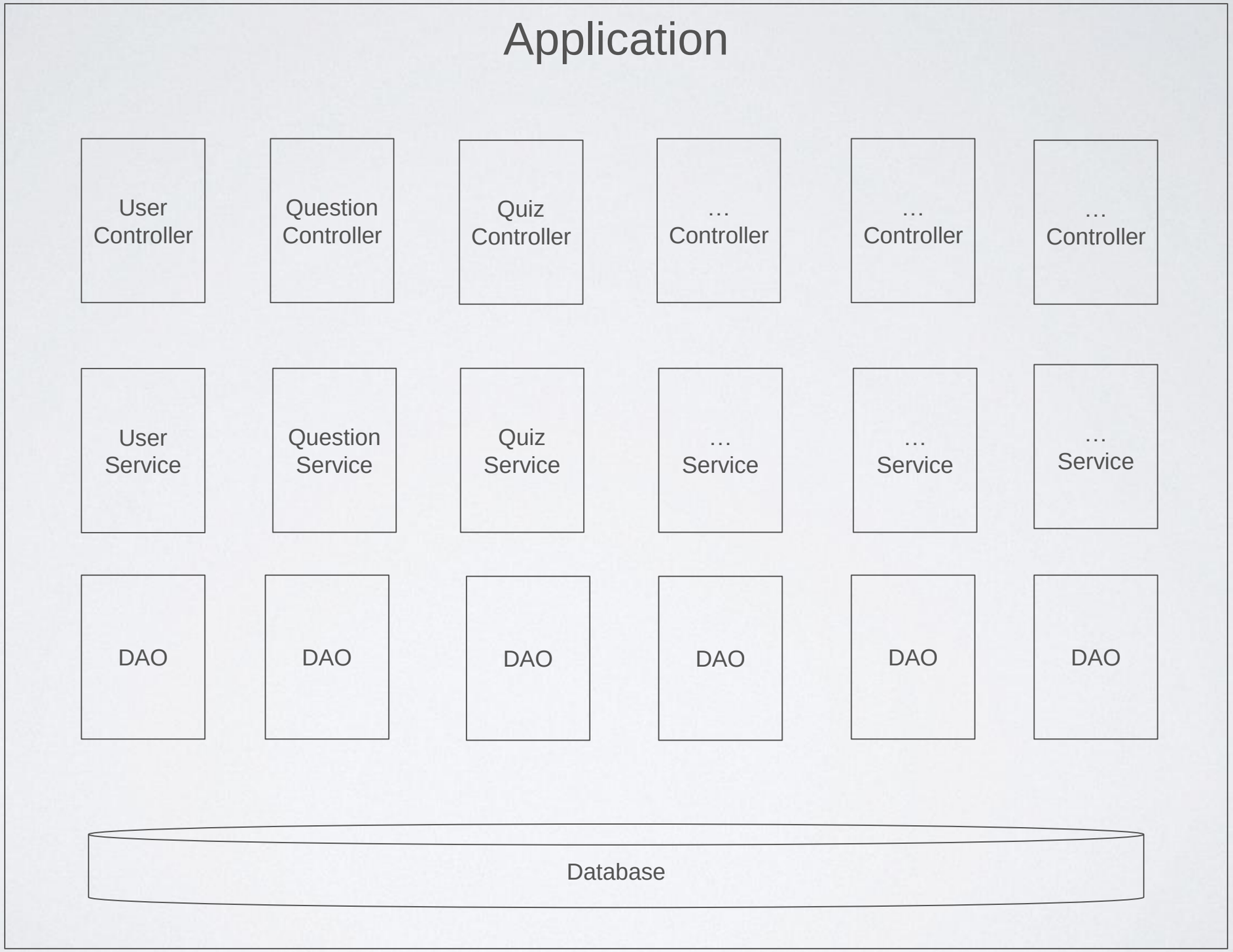
SPRING AOP

- How will you handle following scenarios:
  - For each Service and DAO layer method, your leader wants to know the execution time
  - For each request and response, your leader wants to know what is the request received and what is the response generated
  - If there is any exception thrown during the process of the request, how do you handle those exception?

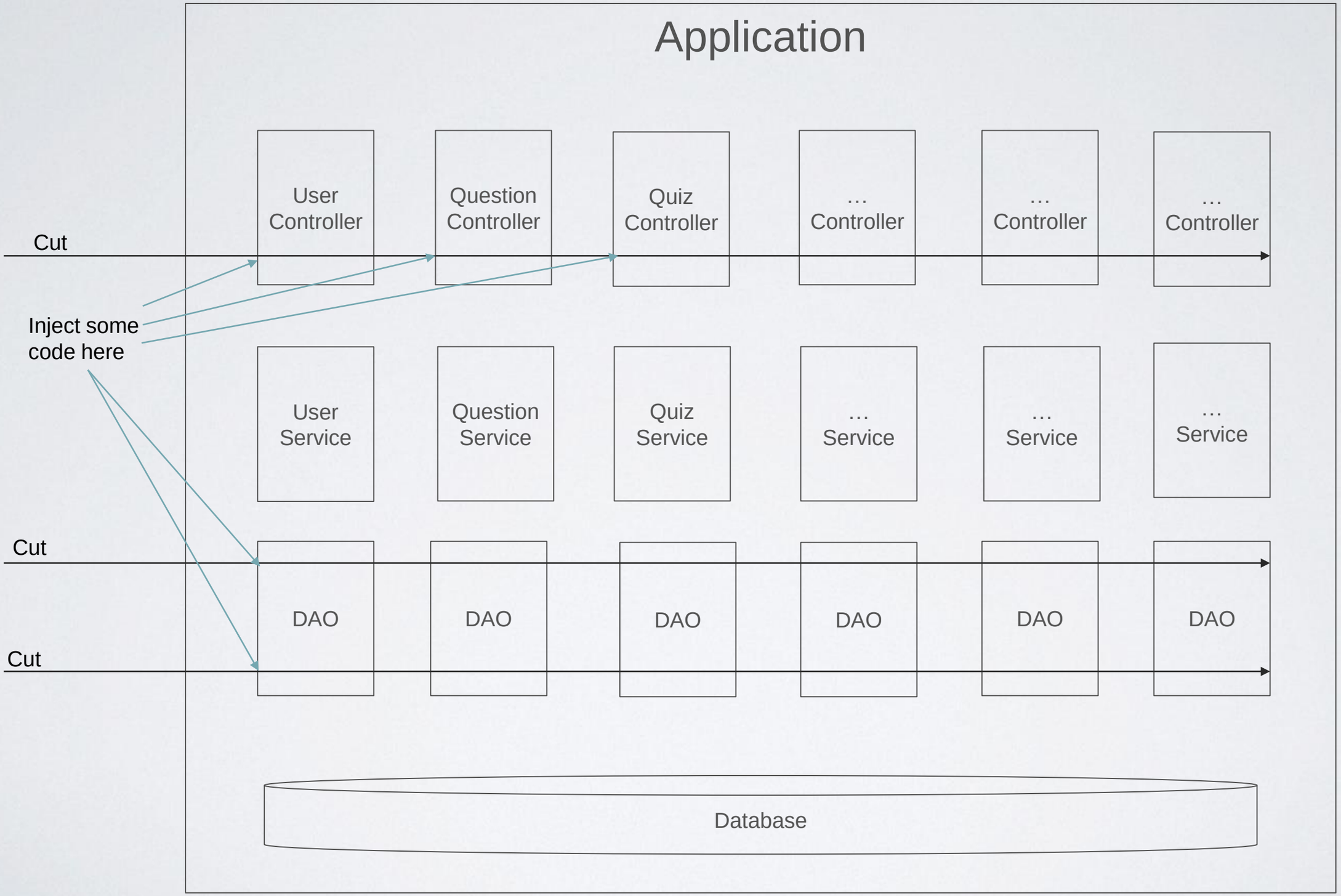
SPRING AOP

- AOP — Aspect-Oriented Programming
  - Aspect-oriented Programming (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 audit log) that cut across multiple types and objects. (Such concerns are often termed “crosscutting” concerns in AOP literature.)

AOP CONCEPT


Application
User Controller
Question Controller
Quiz Controller
… Controller
… Controller
… Controller
User Service
Question Service
Quiz Service
… Service
… Service
… Service
DAO
DAO
DAO
DAO
DAO
DAO
Database

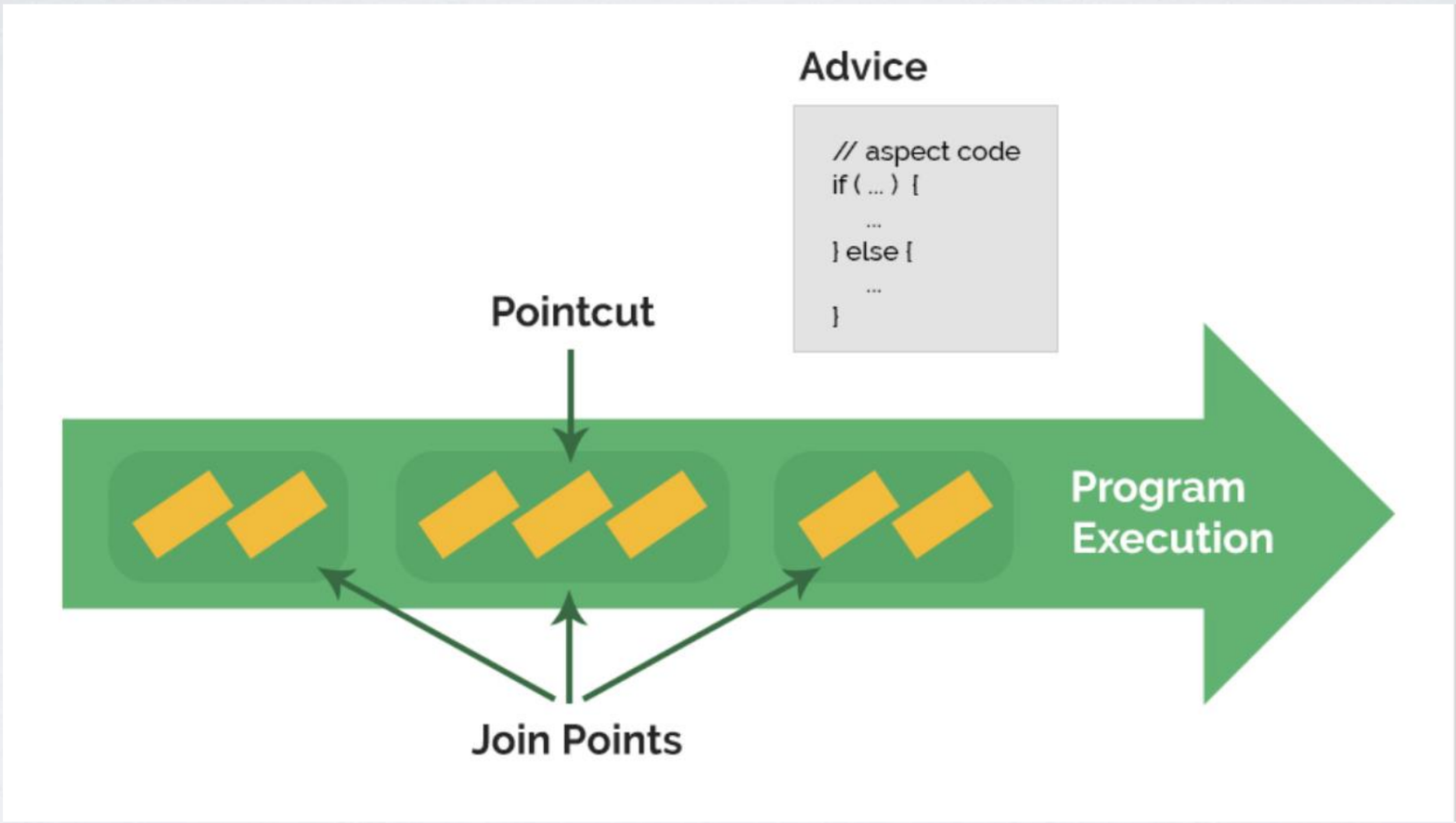
AOP CONCEPT


Application
User Controller
Question Controller
Quiz Controller
… Controller
… Controller
… Controller
Cut
Inject some code here
User Service
Question Service
Quiz Service
… Service
… Service
… Service
Cut
DAO
DAO
DAO
DAO
DAO
DAO
Cut
Database

AOP CONCEPT

- Aspect — A modularization of a concern that cuts across multiple classes
  - Logging / Performance monitoring is a good example of a crosscutting concern in enterprise Java applications
- Join point — A point during the execution of a program, such as the execution of a method or the handling of an exception
- Advice — Action taken by an aspect at a particular join point. Different types of advice include “around”, “before” and “after” advice
- Pointcut — A predicate that matches join points.
- Target object — An object being advised by one or more aspects
- AOP proxy — An object created by the AOP framework in order to implement the aspect contracts
- Weaving — Linking aspects with other application types or objects to create an advised object

AOP CONCEPT



- Instead of directly calling the advice logic inside our business logic, we can define different Join Points by using Pointcut, so that the advice logic will be executed automatically

AOP PROXY

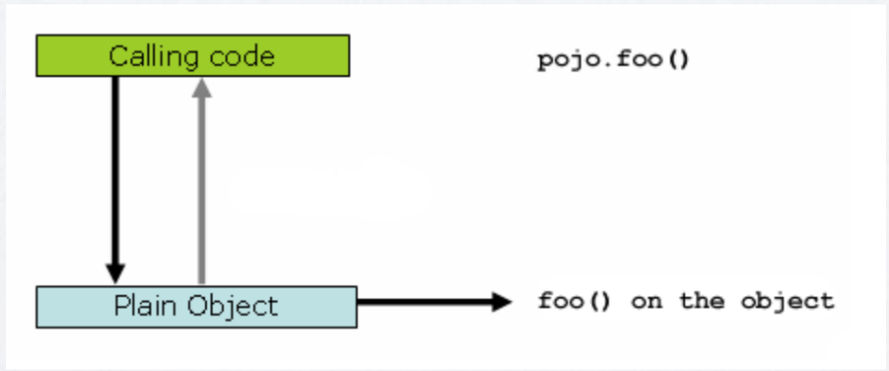
- Spring AOP is proxy-based
- Before we go to proxy, let's take a look at a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this next method invocation is a direct call on the 'this' reference
        this.bar();
    }

    public void bar() {
        // some logic...
    }
}
```

- If you invoke a method on an object reference, the method is invoked directly on that object reference, as the following image and listing show:



AOP PROXY

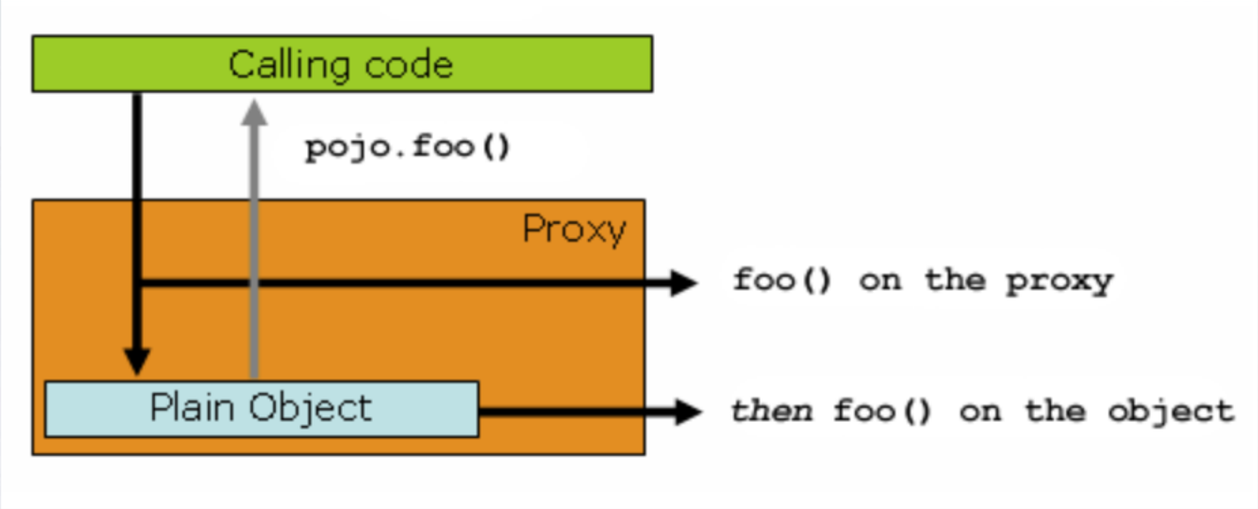
- Things change slightly when the reference that client code has is a proxy.
- Consider the following diagram and code snippet

```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

AOP PROXY



- The key thing to understand here is that the client code inside the main(..) method of the Main class has a reference to the proxy.
- This means that method calls on that object reference are calls on the proxy.
- As a result, the proxy can delegate to all of the interceptors (advice) that are relevant to that particular method call.

AOP PROXY

- How does it work
  - We have a service bean that invokes a call to the saveCustomer( ) method on a DAO
  - We want to have some logging (a cross cutting concern) occur when a call to any save method occurs on a DAO
- Spring detects our need to call on a logging aspect through your AOP configuration or annotations.
  - When it does, it builds a proxy (called CustomerDaoProxy for example sake here) around the "target" object – in this case the DAO
- Now, on a call to a save method in the DAO, the proxy intercepts the call and routes it appropriately to the appropriate advice method in the aspect class

Proxy

- Proxy is a common design pattern for providing a proxy for other objects to control access to an object.
- The proxy class is responsible for preprocessing messages for the delegate class (the proxy class), filtering and forwarding messages, and subsequent processing of messages after they are executed by the delegate class.
- Proxy, in short, both proxy and delegate classes implement the same interface, proxy classes execute methods in delegate classes implemented from the same interface instead of delegate classes, and proxy classes can do other appropriate processing before and after they execute methods instead of delegate classes.

Proxy

- Static Agent
  - Features: Proxy and delegate classes are determined during compilation.
  - **Example**
- Dynamic Agent
  - Dynamic proxy refers to a method by which a client invokes other objects through a proxy class, and dynamically creates a proxy object of the target class when the program runs.
    - JDK dynamic proxy **(Example)**
    - Cglib Dynamic Agent (won't cover here)

SPRING AOP

- Spring provide two ways to configure the AOP
  - AspectJ Supported AOP — Annotation
  - Schema-based AOP — XML (won't cover here)

ASPECTJ

- AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations
- Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching
- The AOP runtime is still pure Spring AOP, though, and there is no dependency on the AspectJ compiler or weaver

@ASPECT

- @Aspect — Aspects can have methods and fields, the same as any other class. They can also contain pointcut, advice

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

- We can register aspect classes as regular beans in your Spring XML configuration or autodetect them through classpath scanning — the same as any other Spring-managed bean.
- However, note that the @Aspect annotation is not sufficient for autodetection in the classpath.
  - For that purpose, you need to add a separate @Component annotation

@POINTCUT

- Pointcuts determine join points of interest and thus enable us to control when advice executes
  - Spring AOP only supports method execution join points for Spring beans, so you can think of a pointcut as matching the execution of methods on Spring beans.
- A pointcut declaration has two parts
  - A signature comprising a name and any parameters
    - A pointcut signature is provided by a regular method definition (the method serving as the pointcut signature must have a void return type)
- A pointcut expression that determines exactly which method executions we are interested in
  - The pointcut expression is indicated by using the @Pointcut annotation

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

@POINTCUT

- Pointcut Designators (PCD)
  - *execution* — For matching method execution join points. This is the primary pointcut designator to use when working with Spring AOP.
  - *within* — Limits matching to join points within certain types (the execution of a method declared within a matching type when using Spring AOP).
  - *this* — Limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type.
  - *target* — Limits matching to join points (the execution of methods when using Spring AOP) 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 when using Spring AOP) where the arguments are instances of the given types.
  - *bean* — Limit the matching of join points to a particular named Spring bean or to a set of named Spring beans (when using wildcards) (Spring AOP Only)

@POINTCUT

- Pointcut Designators (PCD)
  - @*target* — Limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type.
  - @*args* — Limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given types.
  - @*within* — Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP).
  - @*annotation* — Limits matching to join points where the subject of the join point (the method being executed in Spring AOP) has the given annotation.

@POINTCUT

```
@Pointcut("execution(public * (..))")
private void anyPublicOperation() {} 1

@Pointcut("within(com.xyz.someapp.trading..)")
private void inTrading() {} 2

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {} 3
```

- *anyPublicOperation* matches if a method execution join point represents the execution of any public method.
- *inTrading* matches if a method execution is in the trading module.
- *tradingOperation* matches if a method execution represents any public method in the trading module.

@POINTCUT

- The format of an execution expression follows:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)
            throws-pattern?)
```

- All parts except the returning type pattern (ret-type-pattern in the preceding snippet), the name pattern, and the parameters pattern are optional.
- The returning type pattern determines what the return type of the method must be in order for a join point to be matched
- * — It matches any string
- .. — It matches zero or more string
- () — It matches to a method take no parameter

@POINTCUT

```
execution(public * *(..))
```

- The execution of any public method

```
execution(* set*(..))
```

- The execution of any method with a name that begins with set

```
execution(* com.xyz.service.AccountService.*(..))
```

- The execution of any method defined by the AccountService interface

```
execution(* com.xyz.service.*.*(..))
```

- The execution of any method defined in the service package

```
execution(* com.xyz.service..*.*(..))
```

- The execution of any method defined in the service package or one of its sub-packages

```
within(com.xyz.service.*)
```

- Any join point within the service package

```
within(com.xyz.service..*)
```

- Any join point within the service package or one of its sub-packages

@POINTCUT

```
this(com.xyz.service.AccountService)
```

- Any join point where the proxy implements the AccountService interface

```
target(com.xyz.service.AccountService)
```

- Any join point where the target object implements the AccountService interface

```
args(java.io.Serializable)
```

- Any join point that takes a single parameter and where the argument passed at runtime is Serializable

```
@target(org.springframework.transaction.annotation.Transactional)
```

- Any join point where the target object has a @Transactional annotation

```
@within(org.springframework.transaction.annotation.Transactional)
```

- Any join point where the declared type of the target object has an @Transactional annotation

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

- Any join point where the executing method has an @Transactional annotation:

```
bean(tradeService)
```

- Any join point (method execution only in Spring AOP) on a Spring bean named tradeService

ADVICE

- Before advice

```
@Aspect
public class AfterReturningExample {

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

}
```

- After Returning advice

```
@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao..(..))")
    public void doAccessCheck() {
        // ...
    }

}
```

ADVICE

- After Throwing advice

```
@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        // ...
    }

}
```

- After Advice

```
@Aspect
public class AfterFinallyExample {

    @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doReleaseLock() {
        // ...
    }

}
```

ADVICE

- Around advice
  - It has the opportunity to do work both before and after the method executes and to determine when, how, and even if the method actually gets to execute at all.

```
@Aspect
public class AroundExample {

    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
    }

}
```

CONTROLLER ADVICE

- The *@ControllerAdvice* annotation allows us to consolidate our multiple, scattered *@ExceptionHandler*s from before into a single, global error handling component

- It gives us:
  - Full control over the body of the response as well as the status code
  - Mapping of several exceptions to the same method, to be handled together, and
  - It makes good use of the newer RESTful ResposeEntity response

- The order of the *@ExceptionHandlers* does not matter — spring implements a inner priorityqueue to define the order

LOGGING

- When we are working on a large project, we will also hear something below
  - “This feature we deployed last week was working fine till yesterday now I have no idea why is it not working!”
- We can’t be sure how customer will interact with our web application.
  - Even if we have a user friendly guidance, but some behaviors are really hard to expect
  - For example, when the user are calling the customer service to change a plan, and at the same time, he is trying to submit the same order online. It may lead to a failure transaction in our web application. The problem is that when production support team get a call regarding why that online was failed.
  - To figure out what is the issue, we need to have a logging system and keep track of all the operations we made. In such case, when a transaction failed we can check our log and investigate

LOGGING

- There are many logging libraries in Java community, the most commonly used two are :
  - Log4j
  - Logback
- Spring Boot itself will auto configure logging for us. By default it uses Logback.

LOGGING

- There are two commonly used Logging tools:
  - Log4j2

```java
public class HelloWorld {
    private static final Logger logger = LogManager.getLogger("HelloWorld");
    public static void main(String[] args) {
        logger.info("Hello, World!");
    }
}
```

  - Logback

```java
@RestController
public class LoggingController {

    Logger logger = LoggerFactory.getLogger(LoggingController.class);

    @RequestMapping("/")
    public String index() {
        logger.trace("A TRACE Message");
        logger.debug("A DEBUG Message");
```

LOGGING

```
2019-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/7.0.52
2019-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2019-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed
2019-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        : Mapping servlet: 'dispatcherServlet' to [/]
2019-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

- Date and Time: Millisecond precision and easily sortable.
- Log Level: ERROR, WARN, INFO, DEBUG, or TRACE.
- Process ID.
- A --- separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

LOGGING

- Logging Configuration
  - When a file in the classpath has one of the following names, Spring Boot will automatically load it over the default configuration:
    - logback-spring.xml
    - logback.xml
    - logback-spring.groovy
    - logback.groovy

LOGGING

- Logging Configuration
  - Appender — Appenders are responsible for delivering LogEvents to their destination
  - Layouts — An Appender uses a Layout to format a LogEvent into a form that meets the needs of whatever will be consuming the log event
- Pattern — The goal of this class is to format a LogEvent and return the results
  - All pattern conversions can be found at https://logging.apache.org/log4j/2.0/manual/layouts.html#PatternLayout
- Logger — Loggers that are available in the application. Each logger can have multiple appenders.

ANY QUESTIONS?