

Tutorial

Aspect-Oriented Programming

For this tutorial/lab you will walk through a tutorial introducing aspect-oriented programming. You will then complete a lab.

Setup

We will need an IntelliJ project that can use the ajc compiler to go through this Tutorial.

1. **Create an IntelliJ Maven project** that uses the ajc compiler using [this guide](#) from the pre-class assignment.
 - a. Or you can use the Project you created in the pre-class assignment

Aspect

*An aspect acts like a middleman/proxy for objects/methods. **Aspects contain Advices**, methods that contain the code to be injected, **and Pointcuts**, declarations that define where the advice will be injected.*

Advices

*Advices are very similar to a Java method. One of the main differences is that they will only run once a pointcut has been triggered. There are five types of advices: **Before**, **After**, **AfterReturn**, **AfterThrowing**, and **Around**. We will be exploring Before, After, and Around advices.*

Some of the things an advice can do are:

- modify any parameters that are passed into the call.
- modify the return value
- choose to not continue the call to the (method/object) in question
- call other (methods/objects)
- catch any exception that is thrown by the (method/object) in question

Advice Practice

1. **Create a package** in your java folder and name it adviceExample
2. **Download [this MyClass java File](#)** and place it in the adviceExample package
 - a. This is a simple class that just runs a simple `System.out.println()`; when a method is executed
3. **Download this [MyAspect java file](#)** and place it in your adviceExample package

The following is the class declaration that is inside the MyAspect.java file:

```
@Aspect
public class MyAspect {...}
```

This is one of the ways that an aspect can be declared in Java. It is the same as a normal Java class, but it has the “@Aspect” annotation above it.

The following code is inside your public MyAspect class

```
/**
 * This is a pointcut that will only trigger when myMethod method executes
 */
@Pointcut("execution(* adviceExample.MyClass.myMethod(..))")
public void myMethodExecution(){}

/**
 * This is a pointcut that will only trigger when the myMethodAround method executes
 */
@Pointcut("execution(* adviceExample.MyClass.myMethodAround(..))")
public void myMethodAroundExecution(){}
```

These are Pointcut definitions. We will use these pointcuts to execute the different advice functions. For now you do not need to understand the specifics of what the pointcut is doing.

@Before

A Before advice can only modify things before the method/object is called. Everything that runs inside a Before advice runs before the method/object executes.

Look at the following code that is inside the MyAspect(){} class, below the pointcut definitions:

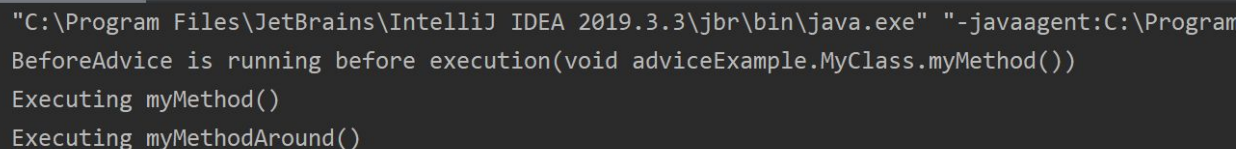
```
/**
 * A Before advice that outputs a string and the method reference
 * @param joinPoint holds a reference of the method call
 */

@Before("myMethodExecution()")
public void beforeAdvice(JoinPoint joinPoint) {
    System.out.println("BeforeAdvice is running before " + joinPoint.toString());
}
```

This Before Advice will execute before every execution of myMethod() in MyClass

1. Every Advice method is marked with @ followed by the advice type. In this instance it is @Before.
2. Inside of every advice annotation, You must place a pointcut that will trigger that advice. In this instance we are using the pointcut we defined as "myMethodExecution()". We defined this pointcut at the top of this file.
3. **The pointcut must be surrounded by quotation marks.** Look at the @Before annotation above for an example.
4. You can use a JoinPoint object in the advice. A **Joinpoint** is a point in your program execution where flow of execution got changed by an aspect, like Exception catching or calling another method.
 - a. A Pointcut is a subset of JoinPoint. A PointCut is just a JoinPoint where we can put our advice.
 - b. A JoinPoint object contains a lot of information, such as parameters that can be used in an advice

Run the MyClass main method. Verify that the method defined in your @Before aspect runs before myMethod() executes. Your console should look like the Screenshot below.



```
"C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\jbr\bin\java.exe" "-javaagent:C:\Program
BeforeAdvice is running before execution(void adviceExample.MyClass.myMethod())
Executing myMethod()
Executing myMethodAround()
```

If it did not work:

1. Verify you are using the ajc compiler, following the [Add the AspectJ ajc compiler to IntelliJ](#) instructions.
2. Try to Re-Build the Project, If you have ajc warnings, Look at what the problem is.

@After

An After advice can only intercept the eventual return value of the method/object that is being called. The @After advice code runs after the method/object executes.

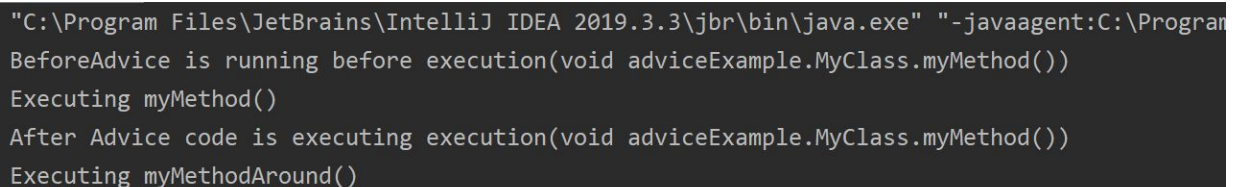
1. **Copy and paste the following code** inside the MyAspect(){} class, below the before Advice.

```
@After("myMethodExecution()")
public void afterAdvice(JoinPoint joinPoint) {
    System.out.println("After Advice code is executing " + joinPoint.toString());
}
```

This After Advice will execute after every execution of myMethod() in MyClass.

2. The @After annotation follows the same rules as @Before, except you replace @Before with @After.
3. The method before the @After annotation will run after the joinpoint finishes executing.

Run the MyClass main method. Verify that the method defined in your @After aspect runs after myMethod() executes. Your console should look like the Screenshot below.



```
"C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\jbr\bin\java.exe" "-javaagent:C:\Program
BeforeAdvice is running before execution(void adviceExample.MyClass.myMethod())
Executing myMethod()
After Advice code is executing execution(void adviceExample.MyClass.myMethod())
Executing myMethodAround()
```

@Around

An around advice is like a combination of Before and After advice. Once it has intercepted a call to the method/object, it has the same control as a Before advice. However, it gets access to any potential return value or exception. It then has the option to continue the operation or to stop it.

1. **Copy and paste the following code** inside the `MyAspect()` class, below the After Advice.

```
/**
 *
 * @param joinPoint the reference to the method call
 * .proceed() signals the method referenced by joinPoint to execute.
 * joinPoint.proceed() returns the return value of the method referenced by joinpoint
 * @return Returns a value in place of the value that is returned by the method held in joinpoint
 * @throws Throwable
 */
@Around("myMethodAroundExecution()")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {

    System.out.println("Around advice is executing before");

    Object returnObject = joinPoint.proceed();

    System.out.println("Around advice is executing After");

    return returnObject;
}
```

This AroundAdvice will print the first line before and the second line after every execution of myMethodAround() in MyClass.

2. The Around Advice uses a `ProceedingJoinPoint` instead of a `JoinPoint`. It extends the `JoinPoint` class and has some extra methods.
3. The `.proceed()` tells the method being targeted by the `PointCut` to execute.
 - a. If you do not tell the `ProceedingJoinPoint` object to `.proceed()` then that method will never execute
4. The Object `returnObject` contains the return value from the method that executed.
 - a. So if the method returns a boolean the object will be a boolean, although you would need to cast it to a boolean before using it as a boolean.

Run the MyClass main method. Verify that the method defined in your @Around aspect runs before and after myMethod() executes. Your console should look like the Screenshot below.

```
"C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\jbr\bin\java.exe" "-javaagent:C:\Program
BeforeAdvice is running before execution(void adviceExample.MyClass.myMethod())
Executing myMethod()
After Advice code is executing execution(void adviceExample.MyClass.myMethod())
Around advice is executing before
Executing myMethodAround()
Around advice is executing After

Process finished with exit code 0
```

Pointcuts

A pointcut defines a location in your code. They are used to tell an advice when to run. There are two main ways to define a pointcut with AspectJ: by annotation, and by pattern. In this tutorial, we will explain definition by annotation and then explain and practice defining pointcuts by pattern.

A pointcut is a subset of JoinPoints. Pointcuts are just JoinPoints that we define so that we can put in advice code.

Note: The ajc will throw an error if you try to execute two advices that are in the same aspect at the same time

Defining pointcuts by annotation

By using annotations, we define pointcuts by manually specifying them with an annotation above the method we want to define as a pointcut. In order to use an annotation, you will need to create a new annotation class.

We will not cover how to use pointcuts by annotation in this tutorial. However the test code you used at the end of the [Create an AspectJ Maven Project guide](#) uses annotation based pointcuts.

Note: An annotation doesn't differentiate between the initialization and execution of a method. So if you mark a pointcut using only an annotation it will run twice. Once when the method is initialized and once when the method executes. This is why the test code uses an annotation and a pattern that looks for all executing methods (to make the pointcut run only once).

Defining pointcuts by pattern

This is the most powerful way to define a pointcut. You can set a pattern for the pointcut to look for and inject the advice code everywhere the pattern matches.

Patterns are designed through the following elements/pattern

[modifiers] [return type] [(packageName) (className) (methodName) (parameters)]

Any one of these elements can use () as a wildcard to accept any input.*

1. [modifiers]
 - a. Can be left blank to be interpreted as *
2. [return type]

- a. void / object / primitive type
- 3. (packageName) (className) (methodName)
 - a. your.package.structure.ClassName.yourMethod -> is valid
 - i. * can be added at any point of the package path to extend the search pattern
- 4. (parameters)
 - a. yourMethod(...) defines any method named yourMethod
 - b. yourMethod(Integer) defines any method named yourMethod that has one parameter that is an Integer

PointCut Practice

1. **Create a new package** in the java folder and name it pointcutExample
2. **Download** the [MyPointcutClass Java file](#) and place it in the pointcutExample Package
 - a. This is a simple class that has 3 different versions of the myMethod() and then 3 other myMethods with an incrementing number identifier at the end.
3. **Download** the [MyOtherClass Java file](#) and place it in the pointcutExample package
 - a. This is a simple class that has 3 different versions of the myMethod()
4. **Download** the [MyPointcutAspect Java file](#) and place it into the pointcutExample package
 - a. This aspect has one Around Advice that we will use to show when certain methods are executed depending on the pointcut that is used
5. **Continue down the tutorial** to practice using the pointcuts given in the MyPointcutAspect Java file.

All Method Executions Pattern

The following code is in your MyPointcutAspect file

```
/**
 * This Pointcut pattern applies to ever method execution in your project
 */
@Pointcut("execution(* *(..))")
public void allMethodExecutions(){}

```

This Pointcut looks for any execution of any method in any class of any package of the project

1. The `allMethodExecutions()` in between the quotes in the `@Around("")` annotation tells the Around advice which pointcut to use

Run the MyPointcutClass main method. Verify that the code defined in your @Around aspect runs before and after every method the main method calls. Your console should look like the Screenshot below.

There will be much more in the console than what is shown on this screenshot

```
"C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\jbr\bin\java.exe" "-javaagent:C:\Program
Before Advice--execution(void pointcutExample.MyPointcutClass.main(String[]))

Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod())
Executing myMethod()
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod())

Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))
Executing myMethod(Integer int), the parameter given is: 5
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))

the inputParameter = A Test String
Before Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
Executing myMethod(String string), the parameter given inside the myMethod is: A Test Strin
After Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
returnString = A Test String
```

All Public Methods Pattern

The following code is in your MyPointcutAspect file below the previous pointcut definition

```
/**
 * This pointcut pattern applies to all public methods in the pointcutExample package
 */
@Pointcut("execution(public * pointcutExample.*(..))")
public void allPublicMethodExecutions(){}


```

This Pointcut looks for any execution of a public method in the pointcutExample package

1. **Copy and paste** `allPublicMethodExecutions()` in between the quotes in the `@Around("")` annotation, replacing what was previously there.

Run the MyPointcutClass main method. Verify that the code defined in your @Around aspect runs before and after every public method that the main method calls. Your console should look like the Screenshot below.


```

Before Advice--execution(void pointcutExample.MyPointcutClass.main(String[]))

Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod())
Executing myMethod()
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod())
|
Executing myMethod(Integer int), the parameter given is: 5

the inputParameter = A Test String
Executing myMethod(String string), the parameter given inside the myMethod is: A Test String
returnString = A Test String

method2

method3

method4

Before Advice--execution(void pointcutExample.MyOtherClass.myMethod())
myOtherClass method
After Advice--execution(void pointcutExample.MyOtherClass.myMethod())

Before Advice--execution(Integer pointcutExample.MyOtherClass.myMethod(Integer))
myOtherClass method with Integer parameter
After Advice--execution(Integer pointcutExample.MyOtherClass.myMethod(Integer))

Before Advice--execution(void pointcutExample.MyOtherClass.myMethod(String))
myOtherClass method with String parameter
After Advice--execution(void pointcutExample.MyOtherClass.myMethod(String))
After Advice--execution(void pointcutExample.MyPointcutClass.main(String[]))

Process finished with exit code 0

```

All method executions with the same parameter inputs

The following code is in your MyPointcutAspect file below the previous pointcut definition

```

/**
 * This pointcut pattern applies to any method named "myMethod" that has a single Integer parameter
 */
@Pointcut("execution(* pointcutExample.MyPointcutClass.myMethod(Integer))")
public void myMethodIntegerExecution(){}

```

This Pointcut looks for the myMethod in the MyPointCutClass class of the pointcutExample package that has an Integer as a parameter

1. **Copy and paste** `myMethodIntegerExecution()` in between the quotes in the `@Around("")` annotation, replacing what was previously there.

Run the `MyPointcutClass` main method. Verify that the code defined in your `@Around` aspect runs before and after every `myMethod(Integer a)` in `MyPointcutClass` that the main method calls. Your console should look like the Screenshot below.

```
Executing myMethod()

Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))
Executing myMethod(Integer int), the parameter given is: 5
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))

the inputParameter = A Test String
Executing myMethod(String string), the parameter given inside the myMethod is: A Test String
returnString = A Test String
|
method2

method3

method4

myOtherClass method

myOtherClass method with Integer parameter

myOtherClass method with String parameter

Process finished with exit code 0
```

All method executions with the same name in the same package

The following code is in your `MyPointcutAspect` file below the previous pointcut definition

```
/**
 * This Pointcut pattern applies to any method in the pointcutExample package that is named "myMethod"
 */
@Pointcut("execution(* pointcutExample.*.myMethod(..))")
public void allMyMethodExecutions(){}


```

This Pointcut looks for any method named `myMethod` in any class of the `pointcutExample` package

1. **Copy and paste** `allMyMethodExecutions()` in between the quotes in the `@Around("")` annotation, replacing what was previously there.

Run the `MyPointcutClass` main method. Verify that the code defined in your `@Around` aspect runs before and after every `myMethod()` in `MyPointcutClass` and `MyOtherClass` that main method executes. Your console should look like the Screenshot below.

```
Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod())
Executing myMethod()
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod())

Before Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))
Executing myMethod(Integer int), the parameter given is: 5
After Advice--execution(void pointcutExample.MyPointcutClass.myMethod(Integer))

the inputParameter = A Test String
Before Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
Executing myMethod(String string), the parameter given inside the myMethod is: A Test String
After Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
returnString = A Test String

method2

method3

method4

Before Advice--execution(void pointcutExample.MyOtherClass.myMethod())
myOtherClass method
After Advice--execution(void pointcutExample.MyOtherClass.myMethod())

Before Advice--execution(Integer pointcutExample.MyOtherClass.myMethod(Integer))
myOtherClass method with Integer parameter
After Advice--execution(Integer pointcutExample.MyOtherClass.myMethod(Integer))

Before Advice--execution(void pointcutExample.MyOtherClass.myMethod(String))
myOtherClass method with String parameter
After Advice--execution(void pointcutExample.MyOtherClass.myMethod(String))
```

All Executions with the same Return Type

The following code is in your `MyPointcutAspect` file below the previous pointcut definition

```
/**
 * This Pointcut pattern applies to any method that returns a String
 */
@Pointcut("execution(String *(..))")
public void allStringReturns(){}


```

This pointcut looks for any method in the package that executes and will return a String

1. **Copy and paste** `allStringReturns()` in between the quotes in the `@Around("")` annotation, replacing what was previously there.

Run the `MyPointcutClass` main method. Verify that the code defined in your `@Around` aspect runs before and after every method that main method executes and return a `String`. Your console should look like the Screenshot below.

```
Executing myMethod()

Executing myMethod(Integer int), the parameter given is: 5

the inputParameter = A Test String
Before Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
Executing myMethod(String string), the parameter given inside the myMethod is: A Test String
After Advice--execution(String pointcutExample.MyPointcutClass.myMethod(String))
returnString = A Test String

method2

method3

method4

myOtherClass method

myOtherClass method with Integer parameter

myOtherClass method with String parameter

Process finished with exit code 0
```

Other Patterns

There are many ways to define a pattern. It would be impossible to review them all.

(Optional) If you want to read more about pointcut definitions you can refer to the official documentation here:

<https://www.eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html>

Lab

In a previous tutorial you learned how to use Prometheus and Grafana to display data. While the data was useful, the code used to initialize the metrics and then collect data at the correct points of the program cluttered the application code. This is an example of code tangling. We will use AOP to separate out all of the Prometheus code from the server operations code.

Setup

1. **Create a new Maven IntelliJ project** using [this guide](#) from the pre-class assignment.
2. **Download these [source files](#)** and place the .java files in the src folder of your IntelliJ project:

Main.java

- The main method:
 - Initializes a thread that runs the serverOperation() method once every second
 - Starts and Runs the algorithm in serverOperation() in a background thread
- The serverOperation() method holds all of the business logic

Dependencies

- **Add the dependencies and Maven compiler plugin [found here](#).**
 - The Maven compiler plugin allows Maven to utilize the Ajc compiler
 - The dependencies allow you to expose data in a format that Prometheus can read

PrometheusAspect

- The PrometheusAspect will contain all of your Prometheus code
 - It contains the following examples
 - **A definition of a Prometheus counter** in an aspect
 - **Two Pointcuts** that target specific methods
 - **An After Advice** that counts the number of times the serverOperations method runs
 - **A Before advice** that creates and runs an HTTPServer which is a Prometheus exporter

BST, BSTNode, and FailedRemoveException

- These are the classes of the Binary Search Tree that the main method will be using.
- **DO NOT MODIFY**
- You do not need to touch these files to instrument your code.

3. Make sure your project's Java Language Level is set up correctly

- Set the Java language level to 8 or later in File -> Project Structure -> Modules
- Navigate to -> **Build,Execution,Deployment | Compiler | Java Compiler**
 - Verify that you have the compiler set to "Ajc"
 - Set the "Target bytecode version" to 1.8 or later

4. Verify that the Lab files are working by running

- If the server started on port 8080 you are ready to begin

- If your server did not start and did not return an error, check that your IntelliJ project is using the Ajc compiler

Requirements

You are to move all of the Prometheus instrumentation from and Main class into the PrometheusAspect class. The modified project must retain the same functionality as the original source files.

You are already given a working example of implementing the numberOfIterations counter metric as an aspect. It is there to help you understand how to structure your PrometheusAspect.

Pointcuts

Hint: [Here is the part of the tutorial that talks about patterns with parameters](#)

1. **Define a pointcut** that looks for when the BST.remove(T data) method is called
 - a. **There are 2 different remove functions in the BST object. Make sure your pointcut is targeting the “public T remove(T data)” method**
 - b. You can use “T” as a Data type when defining a Pointcut
 - i. Ex. `execution(* *.myMethod(T))`
2. **Define a pointcut** that looks for when the BST.add(T data) method is called
 - a. **There are 2 different add functions in the BST object. Make sure your pointcut is targeting the “public boolean add(T data)” method**
 - b. You can use “T” as a Data type when defining a Pointcut
 - i. Ex. `execution(* *.myMethod(T))`

Advices

Hint: [Here is the part of the tutorial that talks about around advice](#)

1. **Create an around advice** that uses the remove pointcut you defined and do the following in the advice:
 - a. Count the number of failed removes

Hint: Use a Try-Catch block in the advice

 - If it throws an exception when you `.proceed()` , then the remove failed
 - If no exception was thrown, then the remove was successful
 - The original Try-Catch block around the remove method in the Main.java will remain, but will no longer have Prometheus code in it.
 - b. Decrement the numberOfNodes gauge when a remove is successful
2. **Create an around advice** that uses the add pointcut you defined and do the following in the advice:
 - a. Count the number of failed adds

- i. Cast the return object retrieved from `joinPoint.proceed()` to a boolean
 1. If that boolean `== false` then the add failed
- b. Increment the `numberOfNodes` gauge when an add is successful
- c. Measure the amount of time it takes for the BST add method to run
 - i. This is the amount of time it takes for the `joinPoint.proceed()` to run, which represents the add functions
 - ii. Start the timer before `joinPoint.proceed()` and observer immediately after `proceed()`;

If you need to touch up on Prometheus Java implementation, [here is the guide](#):

To verify your Aspect Instrumentation:

1. Start Prometheus

- If you no longer have Prometheus installed or no longer remember how to start it, refer to the [Prometheus pre-class assignment](#);
- Download and use this [prometheus.yml](#) for your configuration, it is the same configuration used in the prometheus Lab.

2. Run your program

3. Navigate to <http://localhost:8080/metrics>

- This page will display the raw data being collected by Prometheus from port 8080.
- Data will only be collected while your server is running

Verify Your aspects

If your aspects are created correctly, you should be able to use [this Main.java file](#) in place of your current `Main.java` and have the same functionality that the original lab source `Main.java` file had. Meaning that prometheus is keeping track of:

- The number of failed adds
- The number of failed removes
- The number of nodes currently in the BST tree
- The time it takes for the add method to run

When Finished

1. All the functionality that the original source files had should remain. Except that all of the Prometheus code should have been refactored out into a separate aspect.

What to Turn in

1. Your modified `Main.java`

2. Your modified PrometheusAspect.java