AspectJ Tutorial

AspectJ Cheatsheet

1 Introduction to AspectJ

What is AspectJ?

AspectJ is an implementation of Aspect-Oriented Programming (AOP) for Java. AOP is a programming paradigm that allows you to modularize cross-cutting concerns, which are functionalities that span across multiple parts of your application. It is sort of like a 'Listener' but without having to create an interface for it. Some common examples of cross-cutting concerns are logging, transaction management, security, and performance monitoring.

Key Terminology:

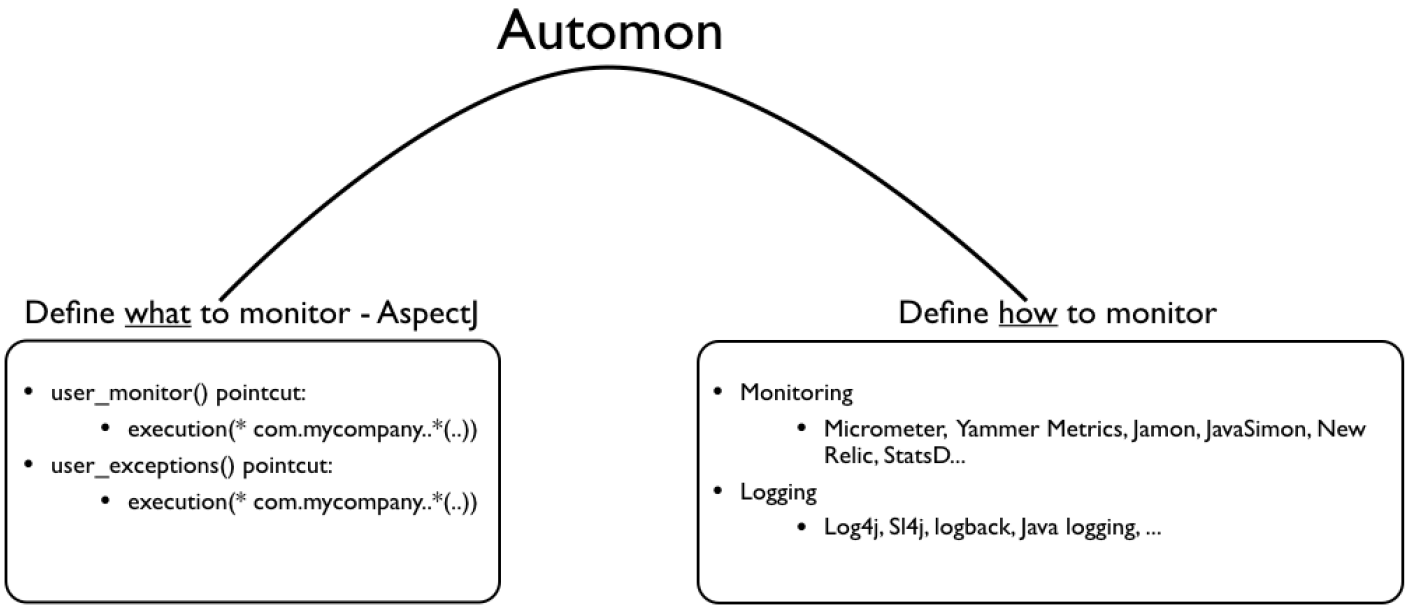
- **Cross-cutting concerns:** Cross-cutting concerns are functionalities that affect multiple parts of a system, making them challenging to modularize with traditional object-oriented programming; aspects provide a way to encapsulate and manage these concerns separately.
- **Aspects:** These are modular units that encapsulate cross-cutting concerns. They define pointcuts, advice, and any additional logic related to the concern.
- **Pointcuts:** These define the specific points in your code where you want to apply the aspect's behavior. They act as filters, identifying join points where the advice should be executed.
- **Advice:** This is the code that is executed at the join points identified by the pointcuts. Advice can be executed before, after, or around the target code.
- **Join points:** These are well-defined points in the execution of your program, such as method calls, field accesses, or exception handling.

- **Weaving:** This is the process of combining aspects with your target code to create the final executable program. Weaving can be done at compile time, load time, or even at runtime.

Use Cases for AspectJ:

- **Logging and tracing:** Capturing logs and traces at specific points in your application to aid in debugging and monitoring.
- **Performance monitoring:** Measuring the execution time of methods or other code blocks to identify performance bottlenecks. (See Automon)
- **Security enhancements:** Enforcing security checks or access control at specific points in your application.
- **Transaction management:** Managing transactions across multiple method calls or database operations.
- **Error handling:** Centralized handling of exceptions and errors across your application
- **Caching:** Improving performance by caching the results of expensive operations.

Performance Monitoring Example (Automon)



2 Core Concepts of AspectJ

Now that we've grasped the fundamentals of AspectJ, let's delve deeper into its core mechanisms: pointcuts and advice.

Pointcuts/Join Points

Pointcuts act as filters, identifying specific points (join points) in your code's execution where you want to apply the aspect's behavior (advice).

Example JoinPoints in a Java Class

Let's illustrate potential join points within a typical Java class to visualize where various pointcut designators can be applied.

```
public class UserService {
    private String databaseUrl; // get(* com.example.service.UserService.databaseUrl), set(* com.example.service.UserService.databaseUrl

    public UserService(String databaseUrl) {  1 usage
        // initialization(com.example.service.UserService.new(..)),
        // execution(com.example.service.UserService.new(..)),
        // constructor call(com.example.service.UserService.new(..)),
        // constructor execution(com.example.service.UserService.new(..))
        this.databaseUrl = databaseUrl;
    }

    static { // staticinitialization(com.example.service.UserService)
        // Initialization code
    }

    private void logUserDetails(User user) { // withincode(* com.example.service.UserService.logUserDetails(..))  1 usage
        // Logging implementation
    }

    public void updateUser(User user) {  1 usage
        // execution(* com.example.service.UserService.updateUser(..)),
        // call(* com.example.service.UserService.updateUser(..)),
        // args(com.example.model.User)
        // Logic to update user in database
    }

    public void deleteUser(int id) throws UserNotFoundException {  1 usage
        // execution(* com.example.service.UserService.deleteUser(..)),
        // call(* com.example.service.UserService.deleteUser(..)),
        // handler(com.example.exception.UserNotFoundException) (if UserNotFoundException is thrown)
        // Logic to delete user from database
    }
```

Some JoinPoints

1. Method-Related Pointcuts
    - execution: Matches the execution of methods based on their signature.
        - Example 1: execution(* com.example.service.*.*(..)) (Matches the execution of any method in any class within the com.example.service package.)
        - Example 2: execution(public String com.example.service.UserService.getUserById(int)) (Matches the execution of the getUserById method in the UserService class.)
    - call: Matches calls to methods based on their signature.
        - Example 1: call(* com.example.repository.*.*(..)) (Matches calls to any method in any class within the com.example.repository package.)
        - Example 2: call(public void com.example.service.OrderService.placeOrder(..)) (Matches calls to the placeOrder method in the OrderService class.)
    - withincode: Matches join points within the body of a specific method.
        - Example 1: withincode(* com.example.controller.UserController.*(..)) (Matches join points within the body of any method in the UserController class.)
        - Example 2: withincode(public void com.example.service.EmailService.sendEmail(String, String, String)) (Matches join points within the body of the sendEmail method in the EmailService class.)
2. Field-Related Pointcuts
    - get: Matches reads (gets) of fields.
        - Example 1: get(* com.example.model.*.*) (Matches reads of any field in any class within the com.example.model package.)
        - Example 2: get(private String com.example.model.User.password) (Matches reads of the password field in the User class.)
    - set: Matches writes (sets) of fields.
        - Example 1: set(* com.example.model.*.*) (Matches writes to any field in any class within the com.example.model package.)

    - Example 2: set(public int com.example.model.Product.quantity) (Matches writes to the quantity field in the Product class.)

3. Constructor-Related Pointcuts

- initialization: Matches the execution of constructors (object initialization).
    - Example 1: initialization(*.new(..)) (Matches the execution of any constructor.)
    - Example 2: initialization(com.example.model.User.new(String, String)) (Matches the execution of the User constructor that takes two String arguments.)
- preinitialization: Matches join points before the execution of constructors (before object initialization).
    - Example 1: preinitialization(*.new(..)) (Matches join points before the execution of any constructor.)
    - Example 2: preinitialization(com.example.model.Order.new(..)) (Matches join points before the execution of any Order constructor.)

4. Type-Related Pointcuts

- within: Matches join points within a specific type (class, interface, or aspect).
    - Example 1: within(com.example.service.*) (Matches join points within any type in the com.example.service package.)
    - Example 2: within(com.example.dao.UserDao) (Matches join points within the UserDao class.)
- this: Matches join points where the currently executing object is of a specific type.
    - Example 1: this(com.example.service.UserService) (Matches join points where the currently executing object is an instance of UserService.)
    - Example 2: this(javax.servlet.http.HttpServletRequest) (Matches join points where the currently executing object is an instance of HttpServletRequest.)
- target: Matches join points where the target object of the join point is of a specific type.
    - Example 1: target(com.example.model.User) (Matches join points where the target object is an instance of User.)
    - Example 2: target(java.util.List) (Matches join points where the target object is an instance of List.)

5. Other Pointcuts

- args: Matches join points based on the arguments passed to a method.

        - Example 1: args(String, int) (Matches join points where the method takes a String and an int as arguments.)
        - Example 2: args(com.example.model.Product) (Matches join points where the method takes a Product object as an argument.)
    - @annotation: Matches join points annotated with a specific annotation.
        - Example 1: @annotation(com.example.annotation.Loggable) (Matches join points annotated with the Loggable annotation.)
        - Example 2: @annotation(org.springframework.transaction.annotation.Transactional) (Matches join points annotated with the Transactional annotation.)
    - handler: Matches the execution of exception handlers.
        - Example 1: handler(java.lang.Exception) (Matches the execution of exception handlers that handle Exception or its subtypes.)
        - Example 2: handler(com.example.exception.CustomException) (Matches the execution of exception handlers that handle the CustomException.)
    - staticinitialization: Matches the execution of static initializers.
        - Example 1: staticinitialization(*) (Matches the execution of any static initializer.)
        - Example 2: staticinitialization(com.example.util.Config) (Matches the execution of the static initializer in the Config class.)

Remember that these are just a few examples. AspectJ provides a rich set of pointcut designators that you can combine and customize to precisely target the join points you need.

Types of Advice

With a solid understanding of pointcuts, let's explore how to define the actions to be taken at these identified join points using advice. AspectJ provides different types of advice, each catering to specific scenarios and behavioral modifications.

- `before`: This advice executes **before** the target code at the join point. It's useful for tasks such as:
    - Logging method entry and parameters
    - Pre-processing data or validating arguments
    - Setting up context or initializing resources

- `after`: This advice executes **after** the target code at the join point, regardless of whether it completes normally or throws an exception. It's suitable for:
    - Logging method exit and return values
    - Cleaning up resources or releasing locks
    - Performing post-processing tasks
- `after returning`: This advice executes **only if the target code completes normally** (without throwing an exception). It's helpful for:
    - Logging successful method execution and results
    - Transforming or modifying return values
    - Performing actions contingent upon successful execution
- `after throwing`: This advice executes **only if the target code throws an exception**. It's ideal for:
    - Logging exceptions and error details
    - Handling or recovering from exceptions
    - Performing compensatory actions
- `around`: This advice **surrounds** the target code, giving you full control over its execution. You can:
    - Proceed with the original execution
    - Modify the arguments or return value
    - Skip the original execution entirely
    - Handle exceptions and retry logic

Accessing Join Point Information

Within the advice body, you can use the `JoinPoint` interface to access valuable information about the join point, including:

- The method being executed (signature, name, declaring type)
- The arguments passed to the method
- The target object on which the method is being invoked
- The return value (if applicable)
- The thrown exception (if applicable)

- Other contextual details

3 Aspects in Action

Having explored pointcuts and advice in theory, let's put them into practice by crafting concrete AspectJ examples. In this section, we'll explore practical AspectJ examples showcasing how to implement common cross-cutting concerns.

Logging Aspect

A logging aspect allows you to capture valuable information about method executions, such as entry/exit points, parameter values, return values, and exceptions. Let's create a simple logging aspect:

```java
@Aspect
public class LoggingAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());  3 usages

    @Pointcut("execution(* com.example.service.*.*(..))")
    public void serviceMethods() {
    }

    @Before("serviceMethods()")
    public void logMethodEntry(JoinPoint joinPoint) {
        logger.info("Entering method: {} with arguments: {}",
                joinPoint.getSignature().toShortString(), joinPoint.getArgs());
    }

    @AfterReturning(pointcut = "serviceMethods()", returning = "result")
    public void logMethodExit(JoinPoint joinPoint, Object result) {
        logger.info("Exiting method: {} with result: {}",
                joinPoint.getSignature().toShortString(), result);
    }

    @AfterThrowing(pointcut = "serviceMethods()", throwing = "exception")
    public void logMethodException(JoinPoint joinPoint, Throwable exception) {
        logger.error("Exception in method: {}: {}",
                joinPoint.getSignature().toShortString(), exception.getMessage());
    }
}
```

Performance Monitoring Aspect

A performance monitoring aspect helps you measure the execution time of methods, aiding in identifying performance bottlenecks:

```java
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Aspect
public class PerformanceMonitoringAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass()); 1 usage

    @Around("execution(* com.example.repository.*.*(..))")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
        long startTime = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long endTime = System.currentTimeMillis();
        long duration = endTime - startTime;
        logger.info("Method {} took {} ms to execute", joinPoint.getSignature().toShortString(), duration);
        return result;
    }
}
```

Exception Handling Aspect

An exception handling aspect provides centralized exception logging and potential retry mechanisms.

Centralized exception handling can streamline error management:

```java
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Aspect
public class ExceptionHandlingAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass()); 1 usage

    @AfterThrowing(pointcut = "execution(* com.example.*.*(..))", throwing = "exception")
    public void logException(JoinPoint joinPoint, Throwable exception) {
        logger.error("Exception in method: {}: {}",
                joinPoint.getSignature().toShortString(), exception.getMessage());
        // You could add retry logic or other exception handling here
    }
}
```

4 Weaving Models in AspectJ

We've seen how aspects interact with your code. Now, let's understand how AspectJ weaves these aspects into your application. Weaving is the process of combining aspects with your target code to create the final executable program. AspectJ offers three primary weaving models:

1. **Compile-time Weaving (CTW)**:

- Aspects are woven directly into your code during compilation.

2. **Post-compile Weaving (PCW)**:

- Aspects are woven into existing class files or JARs after compilation.

3. **Load-time Weaving (LTW)**:

- Aspects are woven as classes are loaded into the JVM.
    i. Example aop.xml - Includes all classes within the com.example package and its subpackages for weaving.

1.
```
<aspectj>
    <!-- Define the aspects to be woven into the application -->
    <aspects>
        <!-- Simple aspect declaration -->
        <aspect name="com.example.aspects.LoggingAspect"/>
        <aspect name="com.example.aspects.PerformanceMonitoringAspect"/>

        <!-- Concrete aspect that extends an abstract aspect -->
        <concrete-aspect name="com.example.aspects.MyConcreteAspect" extends="com.
            <!-- Define a pointcut within the concrete aspect -->
            <pointcut name="myPointcut" expression="execution(* com.example.servic
            <!-- This pointcut matches all method executions in classes under com.
        </concrete-aspect>
    </aspects>

    <!-- Configuration for the AspectJ weaver -->
    <weaver options="-verbose -debug">
        <!-- -verbose: Enables detailed output about the weaving process -->
        <!-- -debug: Provides even more detailed information for debugging purpose

        <!-- Specify which classes should be woven -->
        <include within="com.example..*"/>
        <!-- This includes all classes in com.example package and its subpackages
        <!-- The '..' means "this package and all subpackages" -->
    </weaver>
</aspectj>
```

    ii. Command line to run LTW for the ‘com.example.Main’ program.

```
java -javaagent:path/to/aspectjweaver.jar
 -Dorg.aspectj.weaver.loadtime.configuration=aop.xml
-classpath your-application.jar
com.example.Main
```

5 Advanced AspectJ Features

AspectJ offers advanced capabilities beyond dynamic cross-cutting, allowing compile-time code modifications.

Compile-Time Errors and Warnings

- `declare error`/`declare warning`: Trigger compiler errors/warnings based on conditions in your code, enforcing standards and preventing issues.
    - **Example:** Prevent direct `System.out` calls:

      ```
      declare error: call(* java.io.PrintStream.println(..)) &&
      !within(com.example.logging..*) : "Direct System.out calls are
      prohibited. Use a logger instead.";
      ```
- **Null Argument Checks:** `declare error: call(* *.*(.., null, ..)) : "Null arguments are not allowed.";`
- **Long String Checks:** (Note: This requires runtime checks, not compile-time)
    - `declare warning: args(String s) && if(s.length() > 1000)  : "String argument is too long (over 1000 characters).";`
- **Deprecated Method Calls:** `declare warning: call(@Deprecated * *(..)) : "Calling deprecated method. Consider using an alternative.";`
- **Direct File I/O:** `declare error: call(* java.io.File.*(..)) && !within(com.example.io..*)  : "Direct file I/O is prohibited. Use the FileManager class.";`

Changing Code Structure

- Extending/Enhancing Classes:
  - `declare parents`: Dynamically modify class inheritance (implement interfaces, extend superclasses), but you need to provide the implementation.
  - **Example:** Add an interface: Provide implementation within the aspect or in the target classes themselves
    - `declare parents: com.example.domain.* implements Auditable;`
- Adding Annotations
  - `declare @type`, `declare @method`, etc.: Introduce annotations to classes, methods, fields, or constructors at compile time.
  - **Example:** Annotate service methods:
    - `declare @method: execution(* com.example.service.*.*(..)) : @Transactional;`
- Implementing Interfaces (Mixins):
  - `declare mixin`: "Mix in" interface implementations into existing classes.
  - **Example:** Add logging:

```
declare mixin: com.example.domain.* : Loggable;

interface Loggable {
    void log(String message);
}

aspect LoggingMixin implements Loggable {
    public void log(String message) {
        // Logging implementation
    }
}
```

- Implementing Interfaces (Audit Mixins):
  - Example: Add Auditing

```
declare parents: com.example.domain.* implements Auditable;

interface Auditable {
    void audit();
}

aspect AuditAspect {
    public void com.example.domain.*.audit() {
        // Default audit implementation
        System.out.println("Auditing " + this.getClass().getName
    }
}
```

- Tracking Last Field Change
  - This aspect adds a lastFieldChange field to MyClass and updates it whenever any field within MyClass is modified

```
aspect LastFieldChangeAspect {

    private static final String LAST_CHANGE_FIELD_NAME = "lastFieldChange

    // Introduce a timestamp field to track last change
    private long MyClass.lastFieldChange;

    // Pointcut to capture field modifications within MyClass
    pointcut fieldSetWithinMyClass() : set(* MyClass.*);

    // After advice to update the timestamp on field modification
    after() : fieldSetWithinMyClass() {
        thisJoinPoint.getTarget().lastFieldChange = System.currentTimeMil
    }
}
```