3 - AspectJ: syntax basics 1

Pointcuts

In AspectJ, pointcuts can be either anonymous or named. Anonymous pointcuts, like anonymous classes, are defined at the place of their usage, such as a part of advice, or at the time of the definition of another pointcut. Named pointcuts are elements that can be referenced from multiple places, making them reusable.

Named pointcuts use the following syntax: `[access specifier] pointcut pointcut-name([args]) : pointcut-definition`

Example:

```
public pointcut accountOperations() : call(* Account.*(..))
```

Wildcards and pointcut operators

Three wildcard notations are available in AspectJ:

- `*` denotes any number of characters except the period
- `..` denotes any number of characters including any number of periods
- `+` denotes any subclass or subinterface of a given type

AspectJ provides a unary negation operator (!) and two binary operators (|| and &&).

Signature syntax

In Java, the classes, interfaces, methods, and fields all have signatures. You use these signatures in pointcuts to specify the places where you want to capture join points.

When we specify patterns that will match these signatures in pointcuts, we refer to them as *signature patterns*.

There are 3 type of signatures patterns:

- Type signature patterns
- Method signature patterns
- Method constructor signature patterns
- Field signature patterns

Type signature patterns

The term type refers to classes, interfaces, primitive types and aspects. A type signature pattern in a pointcut specifies the join points in a type at which you want to perform some crosscutting action.

Wildcards:

- `*` is used to specify a part of the class, interface or package name

- `..` is used to denote all direct and indirect subpackages
- `+` is used to denote a subclass or subinterface

Signature Pattern	Matched Types
`Account`	Type of name `Account`.
`*Account`	Types with a name ending with `Account` such as `SavingsAccount` and `CheckingAccount`.
`java.*.Date`	Type `Date` in any of the direct subpackages of the `java` package, such as `java.util.Date` and `java.sql.Date`.
`java..*`	Any type inside the `java` package or all of its direct subpackages, such as `java.awt` and `java.util`, as well as indirect subpackages, such as `java.awt.event` and `java.util.logging`.
`javax..*Model+`	All the types in the `javax` package or its direct and indirect subpackages that have a name ending in `Model` and their subtypes. This signature would match `TableModel`, `TreeModel`, and so forth, and all their subtypes.

Method signature patterns

Method signature patterns allow the pointcuts to identify call and execution join points in methods that match the signature patterns.

In method signatures the wildcard `..` is used to denote any type and number of arguments taken by a method.

Signature Pattern	Matched Methods
`public void Collection.clear()`	The method `clear()` in the `Collection` class that has `public` access, returns `void`, and takes no arguments.
`public void Account.debit(float) throws InsufficientBalanceException`	The public method `debit()` in the `Account` class that returns `void`, takes a single `float` argument, and declares that it can throw `InsufficientBalanceException`.
`public void Account.set*(*)`	All public methods in the `Account` class with a name starting with `set` and taking a single argument of any type.
`public void Account.*()`	All public methods in the `Account` class that return `void` and take no arguments.
`public * Account.*()`	All public methods in the `Account` class that take no arguments and return any type.
`public * Account.*(..)`	All public methods in the `Account` class taking any number and type of arguments.
`* Account.*(..)`	All methods in the `Account` class. This will even match methods with `private` access.
`!public * Account.*(..)`	All methods with nonpublic access in the `Account` class. This will match the methods with `private`, default, and `protected` access.
`public static void Test.main(String[] args)`	The `static main()` method of a `Test` class with `public` access.
`* Account+.*(..)`	All methods in the `Account` class or its subclasses. This will match any new method introduced in `Account`'s subclasses.
`* java.io.Reader.read(..)`	Any `read()` method in the `Reader` class irrespective of type and number of arguments to the method. In this case, it will match `read()`, `read(char[])`, and `read(char[], int, int)`.
`* java.io.Reader.read(char[],..)`	Any `read()` method in the `Reader` class irrespective of type and number of arguments to the method as long as the first argument type is `char[]`. In this case, it will match `read(char[])` and `read(char[], int, int)`, but not `read()`.
`* javax..*.add*Listener(EventListener+)`	Any method whose name starts with `add` and ends in `Listener` in the `javax` package or any of the direct and indirect subpackages that take one argument of type `EventListener` or its subtype. For example, it will match `TableModel.addTableModelListener(TableModelListener)`.
`* *.*(..) throws RemoteException`	Any method that declares it can throw `RemoteException`.

Constructor signature patterns

Constructor signature patterns allow the pointcuts to identify call and execution join points in constructors that match the signature patterns.

In constructor signatures the wildcard `..` is used to denote any type and number of arguments taken by a constructor.

Signature Pattern	Matched Constructors
`public Account.new()`	A public constructor of the `Account` class taking no arguments.
`public Account.new(int)`	A public constructor of the `Account` class taking a single integer argument.
`public Account.new(..)`	All public constructors of the `Account` class taking any number and type of arguments.
`public Account+.new(..)`	Any public constructor of the `Account` class or its subclasses.
`public *Account.new(..)`	Any public constructor of classes with names ending with `Account`. This will match all the public constructors of the `SavingsAccount` and `CheckingAccount` classes.
`public Account.new(..) throws InvalidAccountNumberException`	Any public constructors of the `Account` class that declare they can throw `InvalidAccountNumberException`.

Field signature patterns

Much like the method signature, the field signature allows you to designate a member field. You can then use the field signatures to capture join points corresponding to read or write access to the specified fields.

Signature Pattern	Matched Fields
`private float Account._balance`	Private field `_balance` of the `Account` class
`* Account.*`	All fields of the `Account` class regardless of an access modifier, type, or name
`!public static * banking..*.*`	All nonpublic `static` fields of `banking` and its direct and indirect subpackages
`public !final *.*`	Nonfinal public fields of any class

Implementing pointcuts

There are two ways that pointcut designators match join points in AspectJ.

The first way captures join points based on the category to which they belong. Join points can be grouped into categories that represent the kind of join points they are, such as method call join points, method execution join points, field get join points, exception handler join points, and so forth.

The pointcuts that map directly to these categories or *kinds* of exposed join points are referred as **kinded** pointcuts.

The second way that pointcut designators match join points is when they are used to capture join points based on matching the circumstances under which they occur, such as **control flow**, **lexical scope**, and **conditional checks**.

These pointcuts capture join points in any category as long as they match the prescribed condition. Some of the pointcuts of this type also allow the collection of context at the captured join points.

Kinded pointcuts

Kinded pointcuts follow a specific syntax to capture each kind of exposed join point in AspectJ.

Join Point Category	Pointcut Syntax
Method execution	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(ConstructorSignature)`
Advice execution	`adviceexecution()`

Control-flow based pointcuts

These pointcuts capture join points based on the control flow of join points captured by another pointcut. The control flow of a join point defines the flow of the program instructions that occur as a result of the invocation of the join point.

Think of control flow as similar to a call stack. For example, the `Account.debit()` method calls `Account.getBalance()` as a part of its execution; the call and the execution of `Account.getBalance()` is said to have occurred in the `Account.debit()` method's control flow, and therefore it has occurred in the control flow of the join point for the method.

In a similar manner, it captures other methods, field access, and exception handler join points within the control flow of the method's join point.

A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts.

The first pointcut is expressed as `cflow(Pointcut)`, and it captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself.

The second pointcut is expressed as `cflowbelow(Pointcut)`, and it excludes the join points in the specified pointcut.

Pointcut	Description
`cflow(call(* Account.debit(..))`	All the join points in the control flow of any `debit()` method in `Account` that is called, including the call to the `debit()` method itself
`cflowbelow(call(* Account.debit(..))`	All the join points in the control flow of any `debit()` method in `Account` that is called, but excluding the call to the `debit()` method itself
`cflow(transactedOperations())`	All the join points in the control flow of the join points captured by the `transactedOperations()` pointcut
`cflowbelow(execution(Account.new(..))`	All the join points in the control flow of any of the `Account`'s constructor execution, excluding the constructor execution itself
`cflow(staticinitializer(BankingDatabase))`	All the join points in the control flow occurring during the class initialization of the `BankingDatabase` class

Lexical-structure based pointcuts

A lexical scope is a segment of source code. It refers to the scope of the code as it was written, as opposed to the scope of the code when it is being executed, which is the dynamic scope.

Lexical-structure based pointcuts capture join points occurring inside a lexical scope of specified classes, aspects, and methods.

There are two pointcuts in this category: `within()` and `withincode()`.

The `within()` pointcuts take the form of `within(TypePattern)` and are used to capture all the join points within the body of the specified classes and aspects, as well as any nested classes.

The `withincode()` pointcuts take the form of either `withincode(MethodSignature)` or `withincode(ConstructorSignature)` and are used to capture all the join points inside a lexical structure of a constructor or a method, including any local classes in them.

One common usage of the `within()` pointcut is to exclude the join points in the aspect itself. For example, the following pointcut excludes the join points corresponding to the calls to all print methods in the `java.io.PrintStream` class that occur inside the `TraceAspect` itself: `call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)`

Pointcut	Natural Language Description
`within(Account)`	Any join point inside the `Account` class's lexical scope
`within(Account+)`	Any join point inside the lexical scope of the `Account` class and its subclasses
`withincode(* Account.debit(..))`	Any join point inside the lexical scope of any `debit()` method of the `Account` class
`withincode(* *Account.get-Balance(..))`	Any join point inside the lexical scope of the `getBalance()` method in classes whose name ends in `Account`

Execution object pointcuts

These pointcuts match the join points based on the types of the objects at execution time. The pointcuts capture join points that match either the type `this`, which is the current object, or the `target` object, which is the object on which the method is being called.

Accordingly, there are two execution object pointcut designators: `this()` and `target()`.

In addition to matching the join points, these pointcuts are used to collect the context at the specified join point.

The `this()` pointcut takes the form `this(Type or ObjectIdentifier)`; it matches all join points that have a this object associated with them that is of the specified type or the specified ObjectIdentifier's type. In other words, if you specify `Type`, it will match the join points where the expression `this instanceof <Type>` is true.

The form of this pointcut that specifies ObjectIdentifier is used to collect the this object. If you need to match without collecting context, you will use the form that uses `Type`, but if you need to collect the context, you will use the form that uses `ObjectIdentifier`.

The `target()` pointcut is similar to the `this()` pointcut, but uses the target of the join point instead of `this`.

The `target()` pointcut is normally used with a method call join point, and the target object is the one on which the method is invoked. A `target()` pointcut takes the form `target(Type or ObjectIdentifier)`.

Note that unlike most other pointcuts you cannot use the `*` or `..` wildcard while specifying a type. You don't need to use the `+` wildcard since subtypes that match are already captured by Java inheritance.

Because static methods do not have the `this` object associated with them, the `this()` pointcut will not match the execution of such a method. Similarly, because static methods are not invoked on a object, the `target()` pointcut will not match calls to such a method.

Pointcut	Natural Language Description
`this(Account)`	All join points where `this` is `instanceof Account`. This will match all join points like methods calls and field assignments where the current execution object is `Account`, or its subclass, for example, `SavingsAccount`.
`target(Account)`	All the join points where the object on which the method called is `instanceof Account`. This will match all join points where the target object is `Account`, or its subclass, for example, `SavingsAccount`.

Argument pointcuts

These pointcuts capture join points based on the argument type of a join point.

For method and constructor join points, the arguments are simply the method and constructor arguments. For exception handler join points, the handled exception object is considered an argument, whereas for field write access join points, the new value to be set is considered the argument for the join point.

Argument-based pointcuts take the form of `args(TypePattern or ObjectIdentifier, ..)`.

Pointcut	Natural Language Description
`args(String,.., int)`	All the join points in all methods where the first argument is of type `String` and the last argument is of type `int`.
`args(Remote-Exception)`	All the join points with a single argument of type `RemoteException`. It would match a method taking a single `RemoteException` argument, a field write access setting a value of type `RemoteException`, or an exception handler of type `RemoteException`.

Conditional check pointcuts

This pointcut captures join points based on some conditional check at the join point. It takes the form of `if(BooleanExpression)`.

Pointcut	Natural Language Description
`if(System.currentTimeMillis() > triggerTime)`	All the join points occurring after the current time has crossed the `triggerTime` value.
`if(circle.getRadius() < 5)`	All the join points where the `circle`'s radius is smaller than 5. The `circle` object must be a context collected by the other parts of the pointcut. See section 3.2.6 for details about the context-collection mechanism.