Aspect Oriented Programming

2018-2019
Course 3

Course 3 Contents

- AspectJ Language:
  - Dynamic behavior: advice syntax
  - Examples

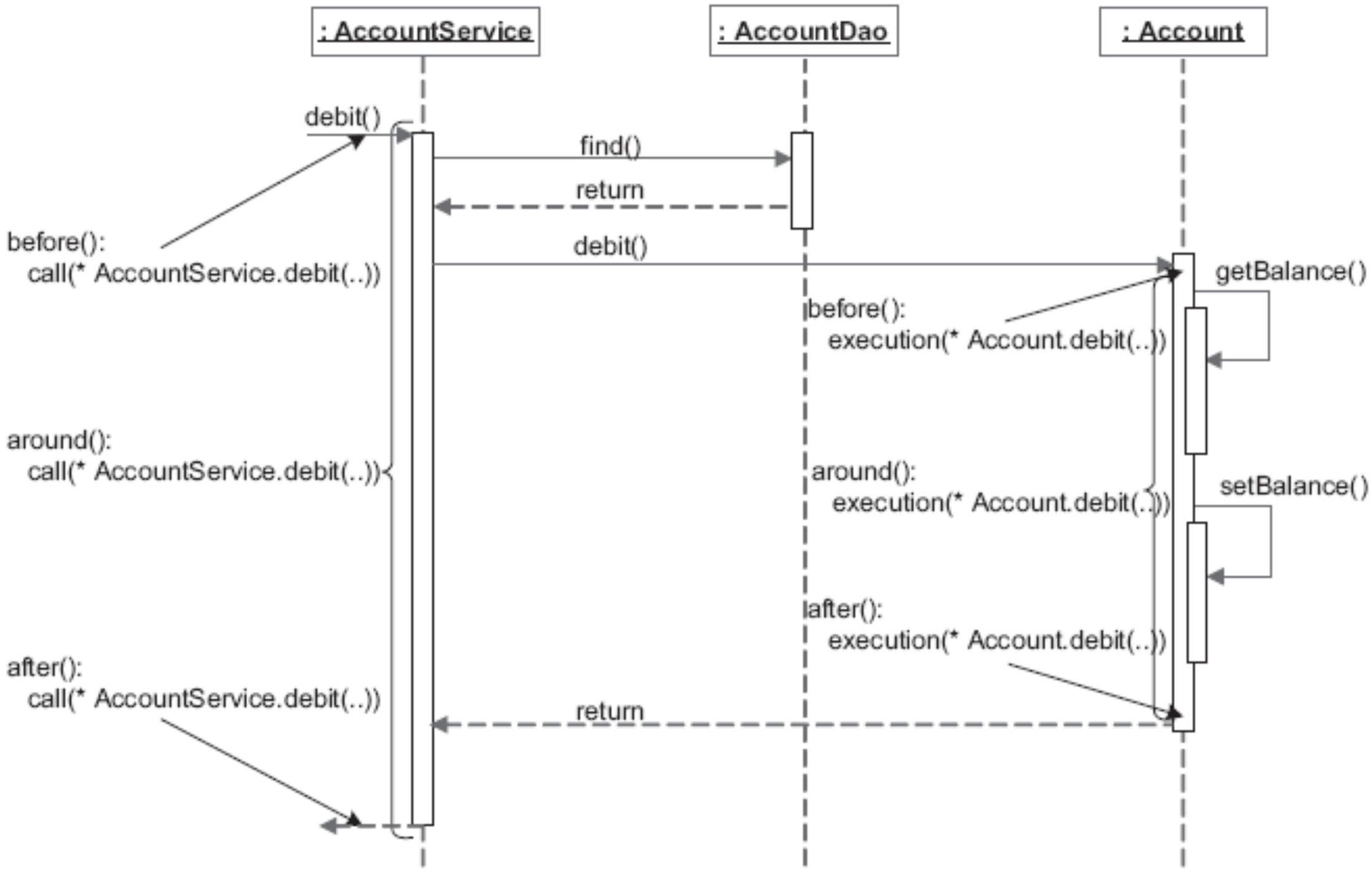
Dynamic Crosscutting

- Dynamic crosscutting constructs provide a way to affect the behavior of a system. The behavior at the join points selected by pointcuts is altered.
- Dynamic crosscutting weaving rules consist of two parts: advice (what to do) and pointcuts (when to apply the advice).
- AspectJ supports dynamic crosscutting through *advice*, a method-like construct, that defines crosscutting action at the join points selected by a pointcut.

Advice Classification

- AspectJ offers three kinds of advice:
  - *Before* advice executes prior to the join point's execution.
  - *After* advice executes following the join point's execution. After advice has three variations based on the outcome of the join point execution:
    - *After* (finally) executes after execution of the join point, regardless of the outcome.
    - *After returning* executes after successful execution of the join point (without throwing an exception).
    - *After throwing* executes after failed execution of the join point (throwing an exception).
- *Around* advice surrounds the join point's execution. It has the ability to continue the original execution with the same or altered context, zero or more times.

Advice Classification



Advice Syntax

- The advice construct has three parts:
  - *Advice declaration*. It specifies if the advice will run before, after, or around the join points.
  - *Pointcut specification.* It specifies which join points will be advised.
  - *Advice body.* It contains the code to execute upon reaching a selected join point.
- Usually, an advice does not have a name, as it is the system's responsibility to execute the advice body upon reaching the selected join points.

Advice Syntax - Example

```
Object around(Connection connection)
    : connectionOperation(connection) {
    long startTime = System.nanoTime();
    Object ret = proceed(connection);
    System.out.println("Operation " + thisJoinPoint
                       + " on " + connection + " took "
                       + (System.nanoTime() - startTime));
    return ret;
}

pointcut connectionOperation(Connection connection)
    : call(* Connection.*(..) throws SQLException)
      && target(connection);
```

1. Advice declaration
2. Pointcut specification
3. Advice body

Advice Syntax - Example

1. The advice declaration specifies when the advice executes relative to the selected join point—before, after, or around it.
    - It may specify the context information available to the advice body (i.e., the execution object and arguments), which the advice body can use to perform its logic
    - It may specify any checked exceptions thrown by the advice.
    - The **`before()`** and **`after()`** advice do not declare a return type, but an **`around()`** advice declares the return type.
2. The pointcut specifies when the advice executes. An advice can be used with any kind of join points exposed by AspectJ.
3. The advice body contains the actions to execute. In **`around`** advice, the **`proceed()`** statement is a special syntax to execute the advised operation.

Comparison Advice-Method

- AspectJ advice syntax is similar to that of methods, but there are also some differences.
- The advice declaration part looks much like a method signature:
  - It optionally has a name (through the @AdviceName annotation).
  - It takes arguments in the form of join point context that the advice body can use to perform its logic, just like in a method.
  - It may declare that it can throw exceptions.
- The advice body looks much like a method body:
  - The code inside the advice body follows the same access-control rules as a method to access members from other types and aspects.
  - Advice can refer to the aspect instance using `this`.
  - An around advice can return a value and therefore declares the return type.
  - Advice must declare checked exceptions that may be thrown by its implementation.

Comparison Advice-Method

- Advice behaves in a similar way to a method overriding the advised join point—it can augment or alter the overridden methods behavior.
- Like overridden methods, an advice:
  - Cannot declare that it may throw a checked exception that is not already declared by all advised join points. The weaver will issue an error if this condition is not met.
  - May omit a few checked exceptions declared by the advised join points.
  - May declare that it can throw more specific checked exceptions than those declared by the advised join points.
  - May throw any runtime exceptions.

Comparison Advice-Method

- Advice differs from methods in several ways.
  - Only optionally has a name.
  - It cannot be called directly. (The system is responsable of executing it.)
  - It does not have an access specifier. (It cannot be directly called anyway.)
  - It does not include a return type in the signature of **`before`** and **`after`** advice.
  - It has access to a few special variables besides **`this`** that carry information about the advised join point: **`thisJoinPoint`**, **`thisJoinPointStaticPart`**, and **`thisEnclosingJoinPointStaticPart`**
  - May use the keyword **`proceed`** in **`around`** advice to proceed with the advised join point.

Before Advice

- Before advice executes before the execution of the advised join point.
- If you throw an exception in the before advice, the advised operation will not execute.
- Before advice is typically used for performing preoperation tasks, such as policy enforcement, tracing, and security.

```
before() : execution(@Secured * *(..)){
      ...authorize user
   }
```

After Advice

- After advice executes after the execution of a join point.
- It is usually important to distinguish between normal returns from a join point and those that throw an exception.
- AspectJ offers three variations of after advice:
  - After, regardless of the outcome
  - After returning normally
  - After throwing an exception

After (Finally)

- The basic form of after advice executes regardless of the join point's outcome: returning normally or throwing an exception.
- It is often referred to as after (finally) advice due to its behavioral resemblance to a finally block, which executes regardless of the outcome of the corresponding try block:

```
after(): call(* Account.*(..)){
     ... log the return from operation
}
```

After Returning

- *after returning* advice is executed after the successful execution of join points.
- If the advised method throws an exception, the advice does not execute.

```
after() returning : call(* Account.*(..)) {
  ... log the successful completion
}
```

- AspectJ offers a variation of the after returning advice that collects the return value. It has the following syntax:

```
after() returning(<ReturnType returnObject>)
```

After Returning

- Although you can modify the collected return object (for example, remove elements from a collection), there is no way to return a new object. If you need such possibility, you must use the around advice.

```
after() returning(Connection conn) :
 call(Connection  DriverManager.getConnection (..)) {
  System.out.println(“Obtained database connection: "+ conn );
}
```

- After advice implicitly limits join point selection to only those that are compatible with the specified return type inside `returning()`.

After Throwing

- After throwing advice executes only when the advised join point throws an exception.
- If a method returns normally, the advice does not execute.

```
after() throwing : execution(* Controller+.*(..)) {
    ... log the failure
}
```

- AspectJ also offers a variation of the after throwing advice that captures the thrown exception object.

```
after() throwing (<ExceptionType exceptionObject>)
```

After Throwing

- This form of the after throwing advice can be used to collect the exception thrown by the advised method so that it can be used in the advice body.

```
after() throwing(Throwable ex ): execution(* *(..)) {
    System.out.println("Exception " + ex + " while executing "
      + thisJoinPoint );
}
```

- The selected join points are implicitly limited to the join points that throw exceptions compatible with the type specified inside `throwing()`.
- Unless the after throwing advice itself throws an exception, the original exception processing continues up the call stack.
- The after throwing advice cannot modify/delete an exception, and the caller of the join point receives the exception thrown by the join point.

Around Advice

- Around advice surrounds join points. It has the ability to execute the join point with the same or different context any number (including zero) of times.
- An around advice may bypass the advised join point or execute the advised join point multiple times, each with different context.
- Some typical usages of around advice are as follows:
  - Perform additional logic before and after the advised join point (eg. profiling)
  - Bypass the original operation and perform some alternative logic (eg.caching)
  - Surround the operation with a try/catch block to perform an exception handling policy (for example, transaction management)

Around Advice

- Around advice is the most powerful form in that it can be always used instead of before or after advice.
- It is recommended to use the simplest form of advice appropriate for the task (in order to precisely express programming intent) .
- Around advice has explicit control over the advised join point's execution.

Proceeding with Advised Join Point

- If within the around advice you want to execute the advised join point, you must use a special keyword—`proceed()`—in the body of the advice.
- Unless you call `proceed()`, the advised join point is bypassed.
- When using `proceed(),` you can pass the context collected by the advice, if any, as the arguments; or you can pass a different set of arguments.
- You must pass the same number and types of arguments as collected by the advice.
- Because `proceed()` causes the execution of the advised join point, it returns the same value returned by the advised join point.
- For example, advising a method that returns a `float` value, invoking `proceed()` returns the same `float` value as the advised method.

Proceeding with Advised Join Point

```
void around(Account account, float amount) throws
  InsufficientBalanceException
: call(void Account.debit(float) throws InsufficientBalanceException)
&& target(account)&& args(amount) {
   try {
     proceed(account, amount);
   } catch (InsufficientBalanceException ex) {
       if (!processOverdraft(account, amount)) {
           throw ex;
       }
   }
}
```

Returning a Value from Around Advice

- Each around advice must declare a return value (which can be void). It's typical to declare the return type to match the return type of the advised join points.

Eg. If you advise a set of methods that are returning an integer, you declare the advice to return an int. For a field-read join point, you match the advice's return type to the accessed field's type.

- In some cases, an around advice applies to join points with different return types. To resolve such situations, the around advice may declare its return value as `Object`.

Returning a Value from Around Advice

- AspectJ resolves return types in the following manner:
  - If a join point returns a primitive type, AspectJ boxes it in its corresponding wrapper type and performs the opposite, unboxing after returning from the advice.
  - If a join point returns a non-primitive type, AspectJ performs appropriate typecasts before assigning the return value. The scheme of returning the `Object` type works even when a captured join point returns a void type.
- The AspectJ weaver issues an error if you specify a return type that is not compatible with any of the advised join points.
- Invoking `proceed()` returns the value returned by the join point. Unless you need to manipulate the returned value, you can return the value that was returned by the `proceed()` statement.
- If you do not invoke `proceed()`, you must still return an appropriate value.

Throwing an Exception from Around Advice

- An around advice, like other advice, must declare any checked exceptions it throws during advice execution.
- Any exceptions thrown by calling `proceed()` need not be declared.

```
void around()
: call(void Account.debit(float) throws InsufficientBalanceException) {
    long start = System.nanoTime();
    proceed();
    long end = System.nanoTime();
    System.out.println("The debit method took "+ (end-start) +
      " nanoseconds");
}
```

Throwing An Exception From Around Advice

- There are situations when the selected join points throw a variety of exceptions, and an **`around()`** advice may need to catch the thrown exception and rethrow the caught exception after taking some action.
- In these cases, you need to use other techniques, including throwing a runtime exception or wrapping the original exception.

Example: Implementing Fault Tolerance

- In a distributed environment, dealing with a network failure is often an important task. If the network is down, clients often reattempt operations.
- The example shows how an aspect with around advice can implement the functionality to handle a network failure.
- 3 classes:
  - RemoteService
  - RemoteClient
  - AOPRemoteException
- 1 aspect
  - FailureHandlingAspect
- Project eclipse

Collecting Join Point Context

- There are situations when pointcuts need to expose the context (eg. method and objects involved) at the point of execution to pass it to the advice implementation.
- Join point context comes in two forms:
  - Objects (including primitives) involved at the join point.
  - Annotations associated with the join point.
- AspectJ provides the `this()`, `target()`, and `args()` pointcuts to collect the objects at the advised join points.
- It provides `@this()`, `@target()`, `@args()`, `@annotation()`, `@within()`, and `@withincode()` pointcuts to collect annotations associated with the advised join points.

Collecting Join Point Context

- You can specify each of the context-collecting pointcuts in two ways:
  - Using the type of the objects
  - Using `ObjectIdentifier`, which is the name of the object.
- When an advice needs the join point context, you use a pointcut that uses the `ObjectIdentifier`.
- Two rules:
  - The advice declares the collected objects in the same way as a method would—each argument specifies a type and name. Then, you use pointcuts to bind each argument to a join point context.
  - The matched pointcuts are implicitly restricted to an equivalent pointcut that uses the type of the object identifier specified.

Collecting Join Point Context

```
before (Account account , float amount ) :
  call (void Account.credit(float))&& target(account) && args(amount) {
     System.out.println("Crediting " + amount + " to " + account );
}
```

- The target object is collected using the `target()` pointcut, whereas the argument value is collected using the `args()` pointcut.
- When named pointcuts are used, those pointcuts must collect the context and pass it to the advice.

```
pointcut creditOperation(Account account, float amount):
call (void Account.credit(float))  && target(account) && args(amount);


before (Account account, float amount) :
          creditOperation(account, amount) {
  System.out.println("Crediting " + amount+ " to " + account );
}
```

Collecting Join Point Context

- AspectJ pointcuts for collecting join point context:
  - `this(obj)` this object at the matching join point.
  - `target(obj)` Target object at the matching join point.
    - For a method call join point, the `target` object is the object on which the method is being invoked.
    - For a method-execution join point, the target object is the `this` object (the same as that collected using `this()).`
    - For field-access join points, the target object is the object whose field is being accessed.
    - For other join points, no target object is available.

Collecting Join Point Context

- **`args(obj1, obj2, ...)`** Objects that represent arguments at the matching join points.
  - For method-call or -execution and constructor-call or -execution join points, it collects the arguments to the method or constructor.
  - For exception-handler join points, it collects the handled exception.
  - For field-modification join points, it collects the new value being set.
- **`@this(annot)`** Annotation associated with the type of the *this* object at the matching join point.
- **`@target(annot)`** Annotation associated with the *target* object.
- **`@args(annot1, annot2, ...)`** Annotations associated with the arguments.
- **`@within(annot)`** Annotation associated with the type enclosing the matching join point.
- **`@withincode(annot)`** Annotation associated with the method enclosing the matching join point.
- **`@annotation(annot)`** Annotation associated with the current join point subject.

Accessing Join Point Context Via Reflection

- AspectJ offers an alternative way to access the static and dynamic context associated with the join points through a reflection API.
- Through this API, you can access the name of the currently advised method as well as the argument objects to that method.
- The most common use of this reflective information is to implement tracing and similar aspects.
- AspectJ provides reflective access to join point context by making three special objects available in each advice body: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`.
- These objects are much like the special variable `this` that is available in each instance method in Java to provide access to the execution object.

Accessing Join Point Context Via Reflection

- The information contained in these three objects is of two types:
  - Static information —It does not change between multiple executions.
    - The name and source location of a method remain the same during different invocations of the method.
  - Dynamic information —Changes with each invocation of the same join point.
    - Two different calls to the method `Account.debit()` will probably have different account objects and debit amounts.
- Each join point provides one object that contains dynamic information and two objects that contain static information about the join point and its enclosing join point.

Accessing Join Point Context Via Reflection

- **`thisJoinPoint`**—This object of type **`JoinPoint`** contains the dynamic information of the advised join point.
  - It gives access to the target object, the execution object, and the method arguments. Through these objects, you can access annotations associated with their types as well.
  - It also provides access to the static information for the join point, using the **`getStaticPart`**() method.
  - You use **`thisJoinPoint`** when you need dynamic information related to the join point (tracing).
- **`thisJoinPointStaticPart`**—This object of type **`JoinPoint.StaticPart`** contains the static information about the advised join point.
  - It gives access to the source location, the kind (method-call, method-execution, field-set, field-get, etc.), and the signature of the join point.
  - You use **`thisJoinPointStaticPart`** when you need the structural context of the join point, such as its name, kind, source location, associated annotations, etc (to log the name of the advised method, but not the parameters).

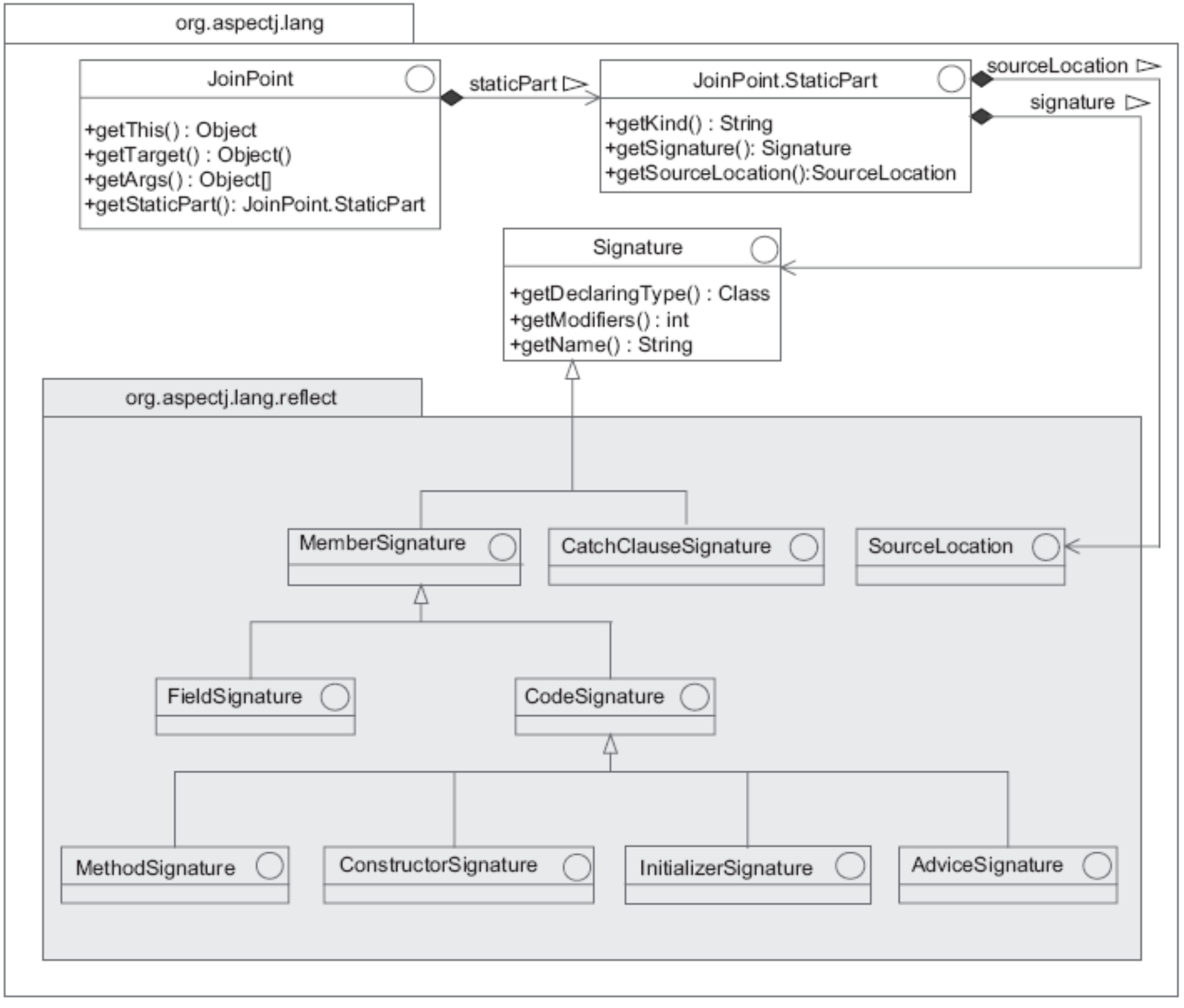
Accessing Join Point Context Via Reflection

- **`thisEnclosingJoinPointStaticPart`**—This object of type **`JoinPoint.StaticPart`** contains the static information about the enclosing join point, which is also referred to as the enclosing context.

- The enclosing context of a join point depends on the kind of join point.
  - for a method-call join point, the enclosing join point is the execution of the caller method.
  - for an exception-handler join point, the enclosing join point is the method that surrounds the catch block.
  - You use the **`thisEnclosingJoinPointStaticPart`** object when you need the context information of the join point's enclosing context (eg. when logging an exception, to log also the enclosing context information).

The Reflection API

- The reflection API in AspectJ is a set of interfaces that together form the programmatic access to the join point information.
- These interfaces provide access to dynamic information, static information, and various join point signatures.
- `org.aspectj.lang` and `org.aspectj.lang.reflect` provide support for accessing the join point information.


org.aspectj.lang
JoinPoint
+getThis() : Object
+getTarget() : Object()
+getArgs() : Object[]
+getStaticPart(): JoinPoint.StaticPart
staticPart
JoinPoint.StaticPart
+getKind() : String
+getSignature(): Signature
+getSourceLocation():SourceLocation
sourceLocation
signature
Signature
+getDeclaringType() : Class
+getModifiers() : int
+getName() : String
org.aspectj.lang.reflect
MemberSignature
CatchClauseSignature
SourceLocation
FieldSignature
CodeSignature
MethodSignature
ConstructorSignature
InitializerSignature
AdviceSignature

Pointcuts vs Reflection API

- An alternative to using the reflection API to collect dynamic context is to use pointcuts provided for that specific purpose (`this(), target(), @this(),` and `@target()).`
- Using reflection to obtain the dynamic context has some disadvantages:
  - It has poorer performance.
  - It lacks static type checking.
  - It is difficult to use.
- Sometimes you need to use reflection because little information is available or required about the advised join points.

Pointcuts vs Reflection API

- Examples with both styles:

```
1.    pointcut pc(Account acc) : this(acc); //pointcut
Account acc = (Account)thisJoinPoint.getThis(); //reflection
```

```
2.    pointcut pc(Account acc) : target(acc); //pointcut
Account acc = (Account)thisJoinPoint.getTarget() //reflection
```

```
3.    pointcut pc(Account acc,Customer cust): args(acc, cust);
 Object[] arguments = thisJoinPoint.getArgs();
Account acc = (Account)arguments[0];
Customer cust = (Customer)arguments[1];
```

Examples

- Logging with AspectJ
- Caching with AspectJ