

Aspect Oriented Programming

2013-2014

Course 3

Course 3 Contents

- ◆ AspectJ Language:
 - The join point model
 - Pointcuts syntax

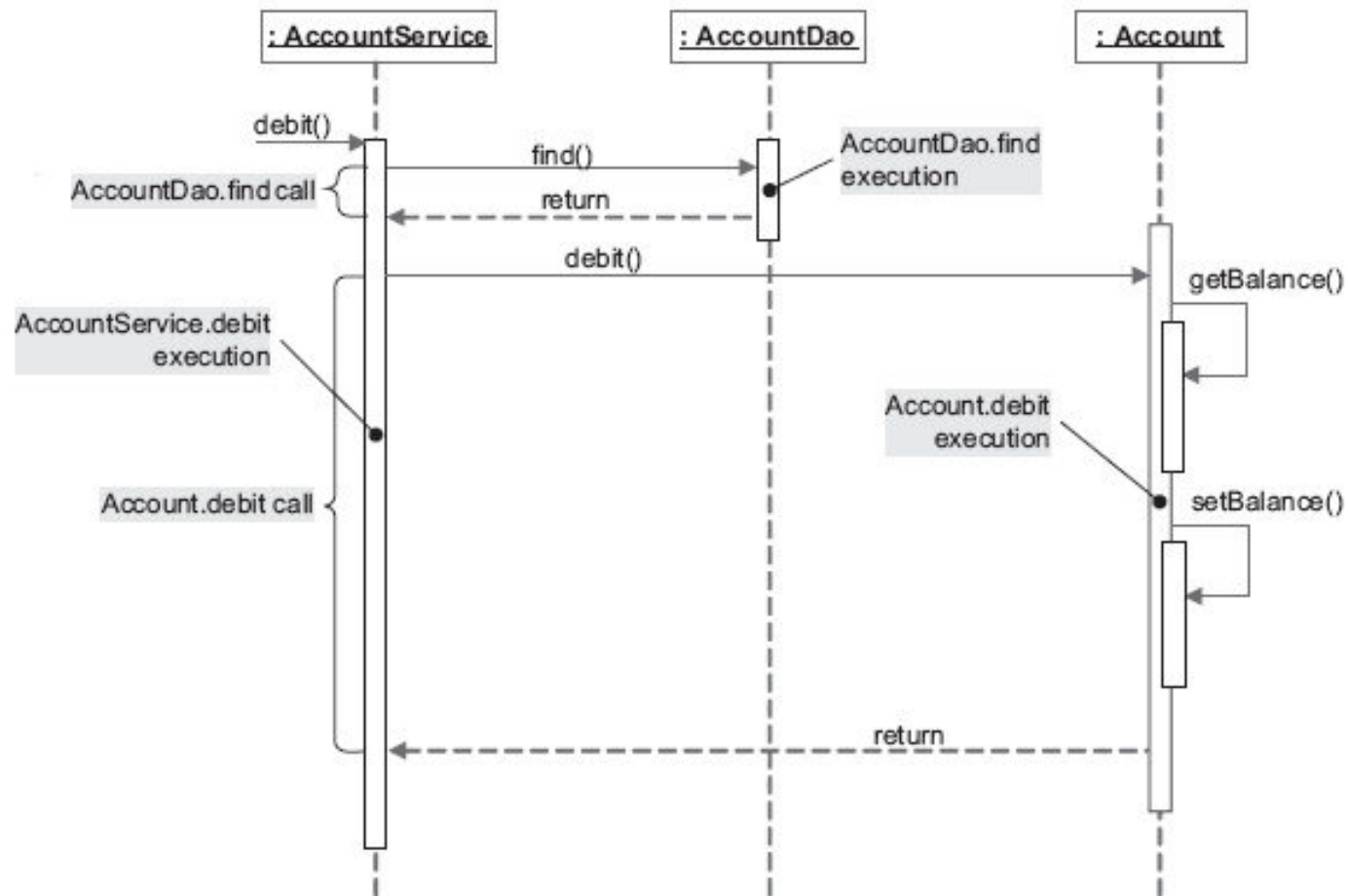
Join point model

- ♦ The join point model is the central concept in AOP.
- ♦ It consists of two parts: *join points*, the points in the execution of an application; and *pointcuts*, a mechanism for selecting join points.
- ♦ AspectJ's pointcut language allows selecting join points based on:
 - structural information such as types, names, arguments, and annotations,
 - runtime conditions such as control flow.

Join points

- ♦ A *join point* is an identifiable execution point in a system (i.e., a call to a method, a field access, a for loop, an if statement, etc.).
- ♦ AspectJ exposes only a subset of all possible join points, such as a method call and a field access, but not for a loop or an if statement.
- ♦ Exposed join points are the only places where we can add crosscutting actions.
- ♦ Method calls and execution are some of the most commonly used join points.

Join points



Pointcuts

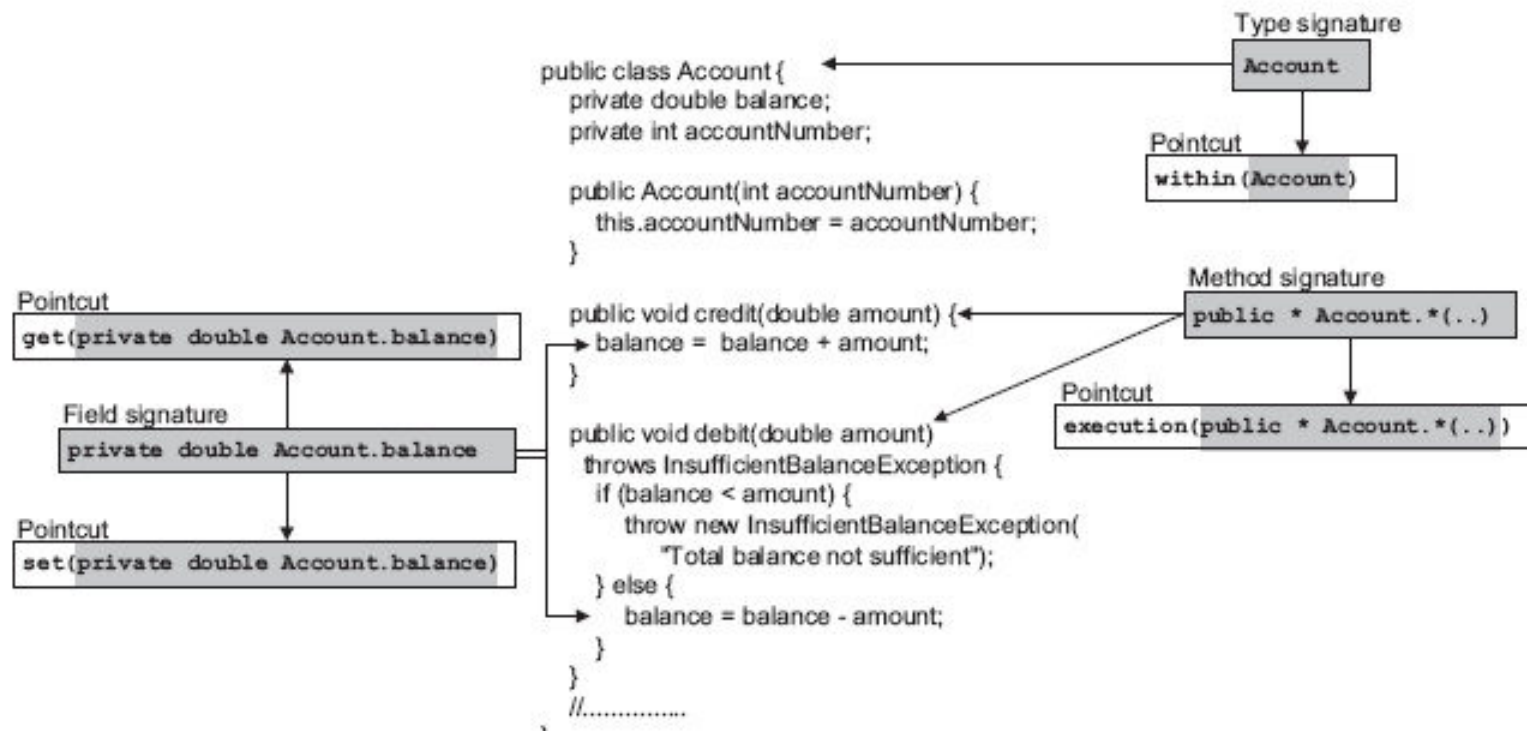
- ◆ A pointcut is a program construct that selects join points and collects join point context.
- ◆ Characteristics:
 - *Selection.* Pointcuts specify a selection criterion. Types, methods, fields, and annotations can be used as the primary mechanism to define pointcuts. Runtime conditions that must be satisfied at a selected join point can also be specified.
 - *Join point context.* A join point has associated runtime information, called join point context, in the form of objects such as the executing object and the method arguments. (Eg. a method call has the caller object, the object on which method is invoked, the arguments, and attached annotations of the method as the join point context).
 - *Signatures.* In Java, all program elements have signatures. Pointcuts use patterns for these signatures to specify the selected join points. Signature patterns can specify wildcards to match a wide range of program elements.

Pointcuts

- *Runtime selection criterion.* AspectJ offers several pointcuts that use runtime information such as the runtime types of the object involved, their values, and the control flow that led to a join point.
- *Determining pointcuts statically.* When we specify a pointcut, the weaver needs to use that information to determine whether a program element matches the specified conditions. For a pointcut that specifies only a structural matching criteria, the weaver can determine the match during the weaving process without any runtime checks. Such pointcuts are called statically determinable pointcuts.

Pointcuts

Relationship between join points, signature patterns, and pointcuts.



AspectJ - Join Points

- ◆ Categories:
 - Method (call/execution)
 - Constructor (call/execution)
 - Field access (read/write access)
 - Exception processing (Handler)
 - Initialization (class initialization, object initialization, object pre-initialization)
 - Advice (execution)

Method join points

- ♦ AspectJ identifies two kinds of join points for methods: *execution* and *call* join points.
- ♦ The *method-execution* join point encompasses the execution of all the code within the body of the method.

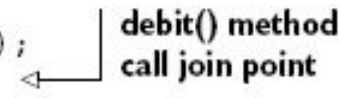
```
public class Account {  
    ...  
    public void debit(double amount)  
        throws InsufficientBalanceException {  
        if (balance < amount) {  
            throw new InsufficientBalanceException(  
                "Total balance not sufficient");  
        } else {  
            balance = balance - amount;  
        }  
    }  
}
```

**debit() method
execution join point**

Method join points

- ◆ The *method-call* join point occurs at the places where a method is being invoked.

```
Account account = new Account(200);  
account.debit(100);
```



debit() method
call join point

- ◆ Remarks:

```
account.debit(Currency.round(100.2345))
```

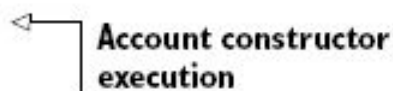
`Currency.round(100.2345)` is not part of the `debit()` method call join point.

- ◆ Execution vs. Call: The difference between the execution and call join points usually does not matter. The most important effect of choosing one over the other relates to weaving.
 - If you advise an execution of a method, the weaver weaves advice into the method body.
 - If you advise calls to a method, the weaver weaves all method invocation locations.

Constructor join points

- ◆ Constructor join points are similar to method join points, except they represent the execution and invocation of object construction.
- ◆ It encompasses the execution of the code within the body of a constructor for an object. If there is not an explicit constructor, the constructor join point exposes the automatically supplied default constructor.
- ◆ The constructor-execution join point represents the entire constructor body.

```
public class Account {  
    ...  
    public Account(int accountNumber) {  
        this.accountNumber = accountNumber;  
    }  
    ...  
}
```



Account constructor execution

The diagram shows a right curly brace on the right side of the code block, spanning the lines of the constructor body. An arrow points from the text 'Account constructor execution' to this brace.

Constructor join points



- ◆ Constructor-call join points represent the points that invoke the creation of an object.

```
Account account = new Account(200);
```

- ◆ Execution vs. Call: Same as for methods join points.

Field access join points


- ◆ Field-access join points correspond to the read and write access to an instance or class member of a class.
- ◆ AspectJ exposes access to instance variables and class variables (static fields) but not to local variables.

```
public class Account {  
    private int accountNumber;  
    private double balance;  
    ...  
  
    public Account(int accountNumber) {  
        this.accountNumber = accountNumber;  Write-access  
join point  
    }  
  
    public String toString() {  
        return "Account: "  
            + accountNumber;  Read-access  
join point  
    }  
    ...  
}
```

Exception-handler join points

- ◆ Exception-handler join points represent the handler block (the catch block) of an exception type .

```
try {  
    account.debit(amount);  
} catch (InsufficientBalanceException ex) {  
    postMessage(ex);  
    overdraftManager.applyOverdraftProtection(account, amount);  
}
```



**Exception
handler
join point**

Class-initialization join points

- ◆ Class-initialization join points represent the loading of a class, including the initialization of the static portion.
- ◆ This join point is present even if there is not an explicit static block; in those cases, it represents the loading of the class, and can be used to weave class load-time actions.


```
public class Account {  
    static {  
        try {  
            System.loadLibrary("accounting");  
        } catch (UnsatisfiedLinkError error) {  
            ... deal with the error  
        }  
    }  
    ...  
}
```

**Account class
initialization**

Object initialization join points

- ◆ Object-initialization join points select the initialization of an object, from the return of a parent class's constructor until the end of the first called constructor. Unlike a constructor-execution join point, it occurs only in the first called constructor for each type in the hierarchy. Unlike class-initialization that occurs when a class loader loads a class, object initialization occurs when an object is created.

```
public class SavingsAccount extends Account {  
    ...  
  
    public SavingsAccount(int accountNumber, boolean isOverdraft) {  
        super(accountNumber);  
        this.isOverdraft = isOverdraft;  
    }  
  
    public SavingsAccount(int accountNumber) {  
        this(accountNumber, false);  
        this.minimumBalance = 25;  
    }  
  
    ...  
}
```

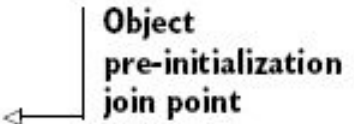


Object initialization
join point

Object pre-initialization join points

- ◆ The object pre-initialization join point is rarely used.
- ◆ It identifies the passage from the first called constructor to the beginning of its parent constructor. (It encompasses calls made while forming arguments to the `super()` call in the constructor.)

```
public class SavingsAccount extends Account {  
    ...  
    public SavingsAccount(int accountNumber) {  
        super(accountNumber,  
              AccountManager.internalId(accountNumber)  
            );  
    }  
    ...  
}
```

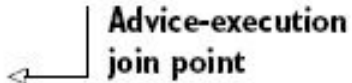


Object
pre-initialization
join point

Advice execution join points

- ♦ AspectJ offers one of its own join points, which encompasses the execution of any advice in the system.
- ♦ Such join points can be advised for purposes such as profiling the advice itself or monitoring executions of advice for unit-testing of aspects.

```
public aspect AccountActivityMonitorAspect {  
    ...  
    before() : accountActivity() {  
        logAccountActivity(thisJoinPoint);  
    }  
}
```



Advice-execution
join point

Join Points - Example

```
public aspect TracingJoinPointsAspect {
    private int indentation;
    pointcut traced() : !(within(TracingJoinPointsAspect) ||
                           withincode(* MessageSender.*(..)));

    before() : traced() {
        print("Before", thisJoinPoint);
        indentation++;
    }
    after() : traced() {
        indentation--;
        print("After", thisJoinPoint);
    }
    private void print(String prefix, Object message) {
        for(int i = 0; i < indentation; i++) {
            System.out.print(" ");
        }
        System.out.println(prefix + ": " + message);
    }
}
```

Join Points - Example

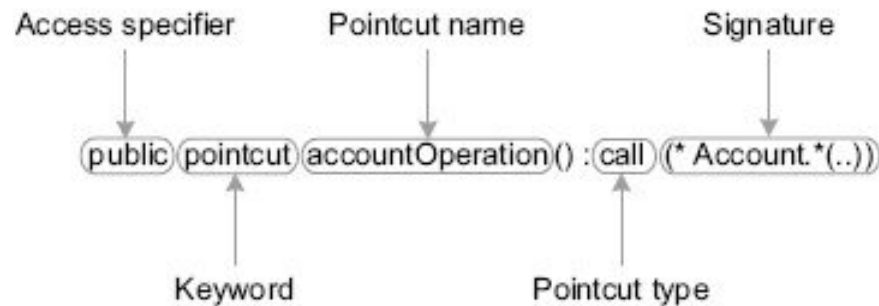
```
//Output for MessageSender.java and Test.java (Course 1)
Before: staticinitialization(Test.<clinit>)
After: staticinitialization(Test.<clinit>)
Before: execution(void Test.main(String[]))
  Before: call(MessageSender())
    Before: staticinitialization(MessageSender.<clinit>)
    After: staticinitialization(MessageSender.<clinit>)
    Before: preinitialization(MessageSender())
    After: preinitialization(MessageSender())
    Before: initialization(MessageSender())
      Before: execution(MessageSender())
      After: execution(MessageSender())
    After: initialization(MessageSender())
  After: call(MessageSender())
  Before: call(void MessageSender.send(String, String, String))
    Before: execution(void MessageSender.send(String, String, String))
After: execution(void MessageSender.send(String, String, String))
  After: call(void MessageSender.send(String, String, String))
  Before: call(void MessageSender.send(String, String))
    Before: execution(void MessageSender.send(String, String))
After: execution(void MessageSender.send(String, String))
  After: call(void MessageSender.send(String, String))
After: execution(void Test.main(String[]))
```

AspectJ Pointcuts

- ◆ A pointcut can be declared inside an aspect, a class, or an interface. An access modifier (private, protected, public or) can be used to restrict access to it.
- ◆ AspectJ pointcuts can be either *anonymous* or *named*.
- ◆ Named pointcuts are elements that can be referenced from multiple places, making them reusable.
- ◆ Anonymous pointcuts, like anonymous classes, are defined at the place of their usage, such as a part of advice, a part of static crosscutting constructs, or when another pointcut is defined.
- ◆ Named pointcuts are recommended and the most commonly used pointcuts.

Named Pointcuts

- ◆ A named pointcut is defined using the **pointcut** keyword and a name. The part after the colon defines the selected join points using the pointcut type and signature.



- ◆ The named pointcut can be used in an advice:

```
before() : accountOperation() {  
    ... advice body  
}
```

Named Pointcuts – Special Form

- ◆ A special form of named pointcut omits the colon and the pointcut definition following it. Such a pointcut selects no join points.
- ◆ Such pointcuts are useful to supply an implementation for an abstract pointcut from a base aspect that selects no join points.
- ◆ It is analogous to implementing a method with an empty body.

```
pointcut requireUpdate(); //no operation requires update
```


Anonymous Pointcuts

- ◆ An anonymous pointcut is defined at the point of its usage.
- ◆ You should avoid using them when the pointcut code is complicated.

```
before() : call(* Account.*(..)) { //anonymous
    ... advice body
}
```

```
pointcut internalAccountOperation() : accountOperation() &&
within(banking..*);
```

- ◆ Named and anonymous pointcuts can be combined using pointcut operators. They allow the creation of complex selection criteria using simpler pointcut definitions.

Pointcut operators

- ♦ AspectJ provides a unary negation operator (!) and two binary operators (|| and &&) to form complex matching rules:
 - Unary operator ! (negation). It matches all join points except those specified by the pointcut.
 - Binary operators (|| and &&). Combining two pointcuts with the || operator selects join points that match either of the pointcuts, whereas combining them with the && operator selects join points matching both pointcuts.
- ♦ The precedence between these operators is the same as in Java.
- ♦ You can use parentheses to override the default operator precedence.

Signature syntax

- ◆ The program-element signature forms the central construct in a pointcut definition.
- ◆ The signature syntax in AspectJ uses wildcards to select program elements that share common characteristics.
- ◆ AspectJ supports three wildcards:
 - ***** denotes any number of characters except the period. In a type signature pattern, it denotes a part of the type or package name. In other patterns, it denotes a part of the name such as the method or field name.
 - **..** denotes any number of characters including any number of periods. In a type signature pattern, it denotes all direct and indirect subpackages. In method signature patterns, it denotes an arbitrary number of method arguments.
 - **+** denotes any subtype of a given type. It may be used only as a suffix to a type signature pattern.

Type signature patterns

- ♦ The term *type* collectively refers to classes, interfaces, annotations, and primitive types.
- ♦ In AspectJ, *type* also refers to aspects. A *type* signature pattern specifies a set of types and may use wildcards, unary, and binary operators.
- ♦ There are 3 types of signature patterns:
 - Basic type signature pattern. It does not use new Java 5 features.
 - Annotation-based type signature pattern. It uses Java 5 annotation feature.
 - Generic-based type signature patterns. It uses Java 5 generics feature.

Basic Type Signature Pattern

- ♦ **Account**

The **Account** type. Only the **Account** type (not its base types or subtypes).

- ♦ ***Account**

Any type with a name ending with **Account**.

eg. **SavingsAccount** and **CheckingAccount**

- ♦ **java.*.Date**

Type **Date** in any of the direct subpackages of the **java** package

eg. **java.util.Date** and **java.sql.Date**

- ♦ **java..***

Any type inside the **java** package or any of its direct and indirect subpackages

eg. Any type in **java.awt** or **java.util**, as well as indirect subpackages (eg **java.awt.event**, **java.util.logging**)

- ♦ **javax..*Model+**

Generic-based Type Signature Patterns

- ◆ `Map<Long, Account>`

The `Map` type, with the first generic parameter bound to the `Long` type and second bound to the `Account` type

eg. `Map<Long, Account>` (but no other base or subtype)

- ◆ `*<Account>`

Any type with `Account` as the parameter

eg. `Collection<Account>`, `Portfolio<Account>`

- ◆ `Collection <? extends Account>`

The `Collection` type, with a type parameter that is `Account` or extends `Account`

eg. `Collection<Account>`, `Collection<SavingsAccount>`,
`Collection<CheckingAccount>`

Combined Type Signatures

- ♦ `!Collection`

Any type other than `Collection`.

eg. `Account`, `ArrayList` (although it extends `Collection`), etc

- ♦ `Collection || Map`

The `Collection` or `Map` type.

eg. `Collection` and `Map` only

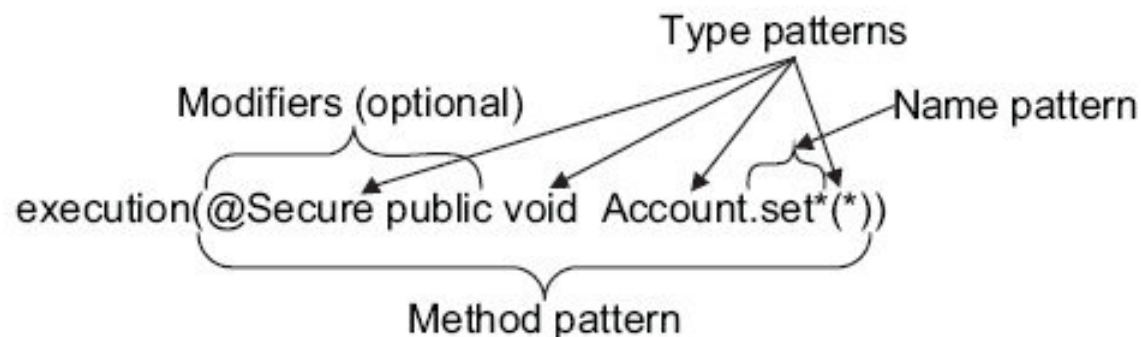
- ♦ `java.util.RandomAccess+ && java.util.List+`

Any type that implements both the specified interfaces.

eg. `java.util.ArrayList`, as it implements both interfaces

Method and Constructor Signature Patterns

- ◆ These signature patterns identify call and execution join points in methods and constructors.
- ◆ Method and constructor signatures specify the name, the return type (for methods only), the declaring type, the argument types, and modifiers.
- ◆ A method pattern uses the type signature patterns extensively.
- ◆ The modifiers (public, private, protected, static, and final, and annotations) are optional, and the matching process ignores the *unspecified* modifiers, if not instructed otherwise.



Examples

- ♦ `public void Account.set* (*)`
Any public method in the `Account` class with the name starting with `set` that returns `void` and takes a single argument of any type.
- ♦ `public void Account.*()`
Any public method in the `Account` class that returns `void` and takes no arguments.
- ♦ `public * Account.*()`
Any public method in the `Account` class that takes no arguments and returns any type.
- ♦ `public * Account.*(..)`
Any public method in the `Account` class that takes any number (including zero) and type of arguments and returns any type.
- ♦ `* Account.*(..)`
Any method in the `Account` class. This matches methods with `public`, `private`, `protected`, and the `default access`.

Examples

- ◆ `* *.*(..)` or `* *(..)`

Any method regardless of return type, defining type, method name, and arguments.

- ◆ `!public * Account.*(..)`

Any method with nonpublic access in the `Account` class. This matches any method with `private`, `protected`, or default access..

- ◆ `* *(..) throws SQLException`

Any method that declares that it can throw `SQLException`.

- ◆ `* Account+.*(..)`

Any method in the `Account` class or its subclasses. This matches any additional method in `Account`'s subclasses.

- ◆ `* java.io.Reader.read (char[], ..)`

Any `read()` method in the `Reader` class, regardless of the type and number of arguments to the method, as long as the first argument type is `char[]` (not `Reader.read()`).

Examples

- ♦ `* 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 takes one argument of type `EventListener` or its subtype..

- ♦ `* java.io.PrintStream.printf(String, Object...)`

The `printf()` method in `java.io.PrintStream` that takes a `String` as the first argument followed by a variable number of `Object` arguments.

... (triple dots) matches how Java specifies variable arguments.

- ♦ `Account AccountService.*(..)`

Any method in `AccountService` that returns the `Account` type. If a subclass overrides the method that declares to return a subtype of `Account` (using the covariant return type feature in Java 5), that is also selected. (There is no need to use the `+` operator).

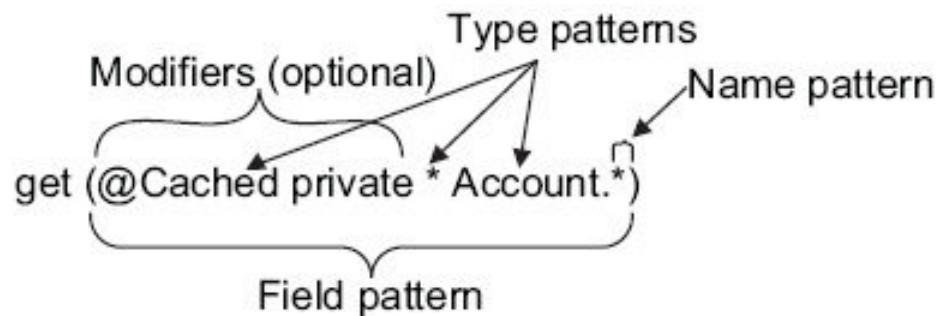
Constructor Signature Pattern

- ◆ A constructor signature differs from a method signature in three ways:
 - It does not allow the return-value specification because constructors do not have a return value,.
 - The method name in a signature is replaced with **new** because constructors do not have names as regular methods do.
 - The **static** keyword cannot be used because constructors cannot be declared static.

The public constructor of the **Student** class, taking no arguments, has the signature **public Student.new()** .

Field Signature Patterns

- ◆ A field signature allows selecting a member field.
- ◆ Field signatures are used to select join points corresponding to read or write access to the specified fields.
- ◆ A field signature must specify the field's type, the declaring type, and the modifiers. Type-signature patterns can be used to specify the types involved.
- ◆ A field-access pointcut selects read (get) or write (set) access to fields matching the specified field signature pattern that, in turn, uses type and name patterns.



Field Signature Patterns

- ◆ Modifiers (access specification, static, and final) are optional. Omitting a modifier selects fields without consideration for the omitted modifier.
- ◆ **private double Account.balance**
Private (instance or static) field balance of the **Account** class.
- ◆ *** Account.***
Any field of the **Account** class, regardless of access modifier, type, or name.
- ◆ *** Account+.***
Any field of the **Account** class or its subclasses, regardless of access modifier, type, or name.

Implementing Pointcuts

- ♦ AspectJ offers several pointcut designators, which when combined with the described signatures form pointcuts.
- ♦ Pointcuts match join points in AspectJ two ways:
- ♦ *Kinded* pointcuts. Pointcuts that directly map to join point categories or kinds, to which they belong (method call/execution, constructor call/execution, field access read/write, etc.).
- ♦ *Non-kinded* pointcuts. Pointcuts that select join points based on information at the join point, such as runtime types of the join point context, control flow, and lexical scope. These pointcuts select join points of any kind as long as they match the condition. Some of the pointcuts of this type also allow the collection of context at the selected join points.

Kinded Pointcuts

- ♦ They follow a specific syntax to select each kind of exposed join point.

Join point category	Pointcut syntax
Method execution	<code>execution (MethodSignature)</code>
Method call	<code>call (MethodSignature)</code>
Constructor execution	<code>execution (ConstructorSignature)</code>
Constructor call	<code>call (ConstructorSignature)</code>
Class initialization	<code>staticinitialization (TypeSignature)</code>
Field read access	<code>get (FieldSignature)</code>
Field write access	<code>set (FieldSignature)</code>
Exception handler execution	<code>handler (TypeSignature)</code>
Object initialization	<code>initialization (ConstructorSignature)</code>
Object pre-initialization	<code>preinitialization (ConstructorSignature)</code>
Advice execution	<code>adviceexecution ()</code>

Kinded Pointcuts - Examples

- ♦ Selecting any public method in the Account class:

```
call(public * Account.*(..))
```

- ♦ Selecting any write access to the balance field of type double and with private access in the Account class:

```
set(private double Account.balance)
```

Kinded Pointcuts Remarks

- ♦ A join point can have only one kind!
- ♦ There cannot be a join point of multiple kinds. Creating a pointcut to match different kinds always leads to no match.

Simple rule: do not combine two pointcuts of different kinds with the **&&** operator.

```
execution(* Account.*(..)) && call(* Account.*(..))
```

“Select join points that match both conditions: it’s an execution of an **Account** method and it’s a call to an **Account** method.”

- ♦ Pointcuts specify a selection criterion that every matching join point must satisfy.

Rule: Use **||** instead of **&&**:

```
execution(* Account.*(..)) || call(* Account.*(..)).
```

“Select join points that match either condition: it’s an execution of an **Account** method or it’s a call to an **Account** method.”

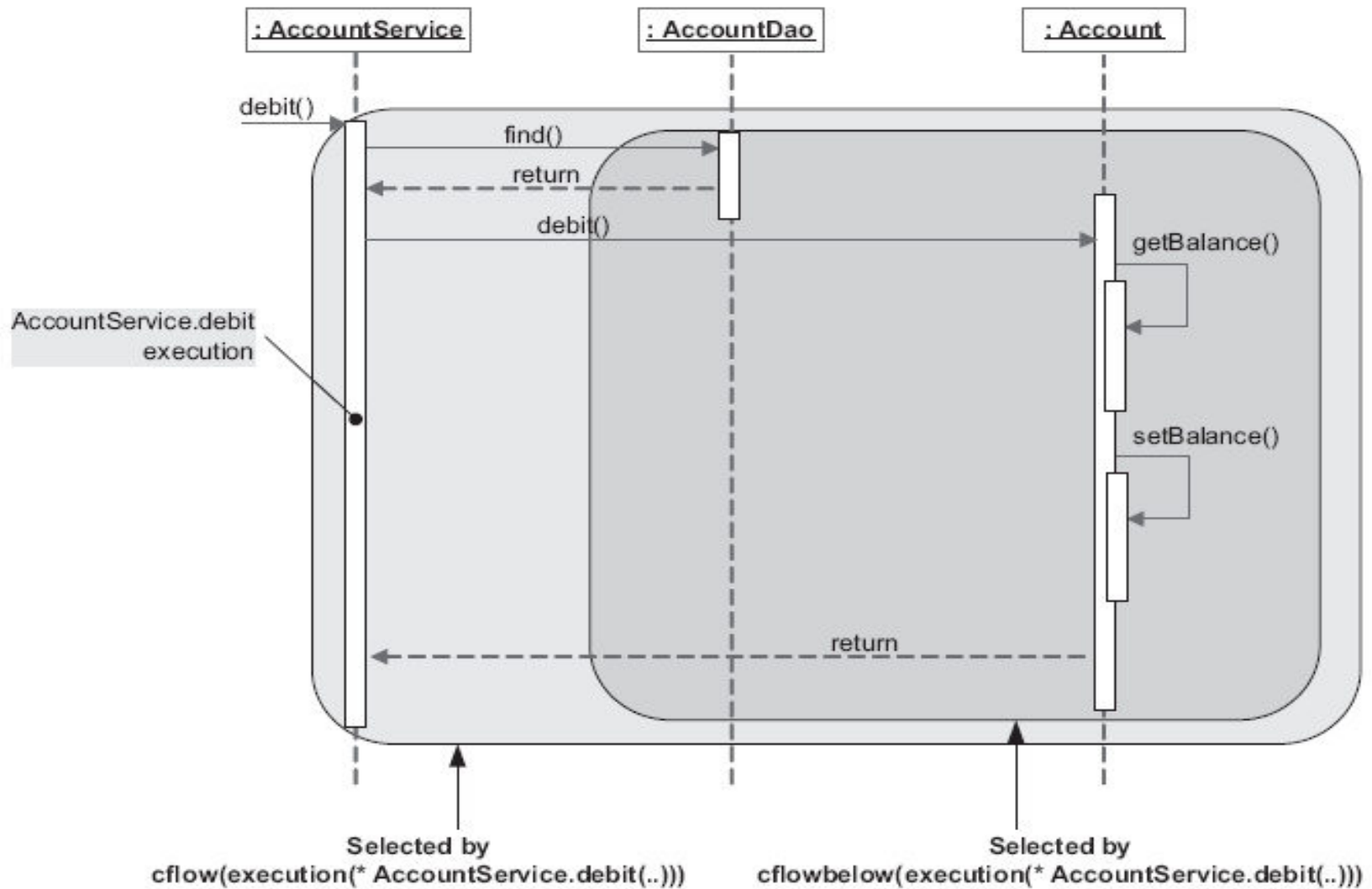
Non-kinded Pointcuts

- ◆ Non-kinded pointcuts select join points based on criteria other than the signature of the join point.
- ◆ For example, with non-kinded pointcuts, you can select all join points occurring inside a class or all join points where the this object is of a certain type. In each case, the selected join points may include method executions, calls, exception handlers, field accesses, and so on, as long as they match the specified criteria.
- ◆ These pointcuts do not consider the kind of join point during selection.
- ◆ AspectJ offers non-kinded pointcuts based on a program's control flow and its lexical structure, execution object, arguments, annotations, and conditional expressions.

Control-flow Based Pointcuts

- ◆ These pointcuts select join points occurring in the control flow of join points selected 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.
- ◆ A control-flow pointcut specifies another pointcut as its argument. There are two control-flow based pointcuts:
 - **cflow (<Pointcut>)**—Selects the join points in the control flow of the specified pointcut, including the join points matching the pointcut
 - **cflowbelow (<Pointcut>)**—Selects the same join points as **cflow()**, except for the join point that initiated the control flow.

Control-flow Based Pointcuts



Control-flow Based Pointcuts – Examples

- ♦ `cflow(execution(* Account.debit(..)))`

All join points in the control flow of execution of any `debit()` method in `Account`, including the execution of the `debit()` method.

- ♦ `cflowbelow(execution(* Account.debit(..)))`

All join points in the control flow of execution of any `debit()` method in `Account`, but excluding the execution of the `debit()` method.

- ♦ `cflow(transacted())`

Any join point in the control flow of the join points selected by the `transacted()` pointcut.

Lexical-structure Based Pointcuts

- ◆ A lexical scope is a segment of source code. Lexical-structure-based pointcuts select join points occurring inside the lexical scope of specified classes, aspects, and methods.
- ◆ Two pointcuts fall into this category:
 - **within(<TypePattern>)**—Selects any join point within the body of the specified classes and aspects, as well as any nested classes.
 - **withincode(<ConstructorSignature>)** or **withincode(<MethodSignature>)**—Selects any join point inside a lexical structure of a constructor or a method, including any local classes in them.

Lexical-structure Based Pointcuts – Examples

- ♦ `within(Account)`

Any join point inside the `Account` class's lexical scope including inside any nested classes.

- ♦ `within(Account+)`

Any join point inside the lexical scope of the `Account` class and its subclasses including any inside nested classes

- ♦ `withincode(* Account.debit(..))`

Any join point inside the lexical scope of any `debit()` method of the `Account` class including inside any local classes.

- ♦ `traced() && !within(TraceAspect)`

One common usage of the `within()` pointcut is to exclude the join points in the aspect. If you advise this pointcut, the `!within()` part ensures that even if the `traced()` pointcut (such as `execution(* *(..))`) selects join points in the aspect, they will not be advised.

Execution-object Pointcuts

- ♦ These pointcuts select join points based on the types of the objects at execution time.
- ♦ The pointcuts select join points that match either the type of *this*, which is the current execution object, or the *target* object, which is the object on which the method is being called.
- ♦ There are two execution object pointcut designators:
 - **this()**—Takes the form **this(<Type or ObjectIdentifier>)**. It selects join points that have a **this** object associated with them that is of the specified type or the specified ObjectIdentifier's type. If you specify **Type**, it matches the join points where the expression **this instanceof <Type>** is true, leading to matching subtypes as well. The form of this pointcut that specifies **ObjectIdentifier** collects the this object so you can use that object in advice.

Execution-object Pointcuts

- **target()**—Similar to the **this()** pointcut but uses the target of the join point instead of this. It takes the **target(<Type or ObjectIdentifier>)** form. 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.
- ◆ These pointcuts can collect the context at the specified join point.
- ◆ **this()** and **target()** pointcuts take only Type as their argument. No wildcard can be used while specifying the type.
- ◆ Due to Java's use of erasure for generics implementation, AspectJ doesn't allow the type pattern for these pointcuts to be a generic type along with its parameters.
- ◆ It is an error to specify **target(List<Account>)**. However, you may specify a generic type without the parameters such **target(List)**.

Execution-object Pointcuts - Examples

- ♦ **this (Account)**

Any join point where **this instanceof Account** evaluates to true. This selects all join points, such as methods calls and field assignments, where the current execution object is **Account** or one of its subclasses: for example, **SavingsAccount**.

- ♦ **target (Account)**

Any join point where the object on which the method is being called is **instanceof Account**. This selects all join points where the target object is **Account** or one of its subclasses: eg. **SavingsAccount**.

Execution-object Pointcuts - Remarks

- ♦ The `this()` pointcut will not select the execution of a static method because it does not have an `this` object associated.
- ♦ The `target()` pointcut will not match calls to a static method, because it is not invoked on an object (usually).
- ♦ Differences in the way `within()` and `this()` perform matching.
 - The former matches when the object in the lexical scope matches the type specified in the pointcut, whereas the latter matches when the current execution object is of a type that is specified in the pointcut or its subclass.
 - The two pointcuts `execution(* Account.*(..))` and `execution(* *.*(..)) && this(Account)` don't select the same set of join points. The first selects all the instance and static methods defined in the `Account` class, whereas the latter picks up the instance methods in the class hierarchy of the `Account` class but none of the static methods.

Execution-object Pointcuts - Remarks

```
public class Account {  
    ...  
    public void debit(double amount)  
        throws InsufficientBalanceException {  
        ...  
    }  
    private static class Helper {  
        ...  
    }  
}  
public class SavingsAccount extends Account {  
    ...  
}
```

**Selected
by within
(Account)**

**Selected
by this
(Account)**



Argument Pointcuts

- ◆ These pointcuts select join points based on the argument object's runtime type of a join point. Objects considered as argument objects differ depending on the join point kind:
 - For method and constructor join points, the argument objects are the method and constructor arguments.
 - For exception-handler join points, the argument object is the handled exception.
 - For field write access join points, the argument object is the new value to be set.
- ◆ Argument-based pointcuts take the form `args (Type or ObjectIdentifier, ..) .`
- ◆ The selection is based on runtime type (similar to the `this()` or `target()` pointcuts) and not the declared type in program element.
- ◆ These pointcuts can be used to collect the context.

Argument Pointcuts - Examples

- ◆ `args(Account, ..., int)`

Any method or constructor join point where the first argument is of type `Account` and the last argument is of type `int`. This matches methods such as `print(Object, float, int)` as long as the first argument passes the `instanceof Account` test.

- ◆ `args(RemoteException)`

Any join points with a single argument of type `RemoteException`. It matches a method or constructor 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 selects join points based on some conditional check at the join point. It takes the form of `if(BooleanExpression)`.
- ◆ The `if()` pointcut is often combined with other pointcuts to selectively apply dynamic crosscutting.

- ◆ `if(debug)`

Any join point where the debug static field (in the defining aspect) is set to true.

- ◆ `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 less than 5. The circle object must be a context collected by the other parts of the pointcut or a static field in the defining aspect.