

# **Introduction to Aspect-Oriented Programming**

Martin Giese

Chalmers University of Technology

Göteborg, Sweden

---

# Advanced AspectJ

# Remaining Topics

---

- advice and exceptions
- exception softening
- abstract aspects
- aspect precedence
- aspect instantiation
- static cross-cutting facilities
- privileged aspects

# Exceptions in Advice

---

Advice body may throw exceptions:

```
before() : doingIO() {  
    openOutputFile();  
}
```

If `openOutputFile()` throws `java.io.IOException`:

```
before() throws java.io.IOException : doingIO() {  
    openOutputFile();  
}
```

▣▣▣▣➡ `IOException` must be declared/handled  
at all places where pointcut applies.

# Reminder: Java checked exceptions

---

Java has *checked* and *unchecked* exceptions.

- Any sub-type of `java.lang.RuntimeException` is unchecked.  
Unchecked exceptions need not be declared in method headers, etc.  
(Also applies to subclasses of `java.lang.Error`)
- Any other sub-type of `java.lang.Exception` is checked. Checked exceptions must either be handled in a method body, or be declared in method headers.

# Aspects throwing exceptions

---

Sometimes, an aspect can change the behaviour of methods, so that new *checked* exceptions are thrown:

- Add synchronization: `InterruptedException`
- Execute calls remotely: `RemoteException`

▣▣▣▣➔ Two possibilities:

- catch and handle exception directly in advice. Might not be appropriate.
- pass exception out of advice. Needs lots of declarations.

# Softening Exceptions

---

Softening an exception makes a checked exception unchecked:

```
declare soft: Type : Pointcut;
```

Exceptions of that type occurring at that pointcut  
are wrapped in an unchecked `org.aspectj.lang.SoftException`  
and re-thrown.

▣▣▣▣➤ Further up in the program catch the `SoftException`

Needs to be used with care, to make sure that all  
softened exceptions do eventually get handled appropriately.

# Softening Example

---

Example from before:

```
before() : doingIO() {  
    openOutputFile();  
}
```

where `openOutputFile()` throws `IOException`.

Add the following:

```
declare soft : java.lang.IOException :  
    call(* *.openOutputFile())
```

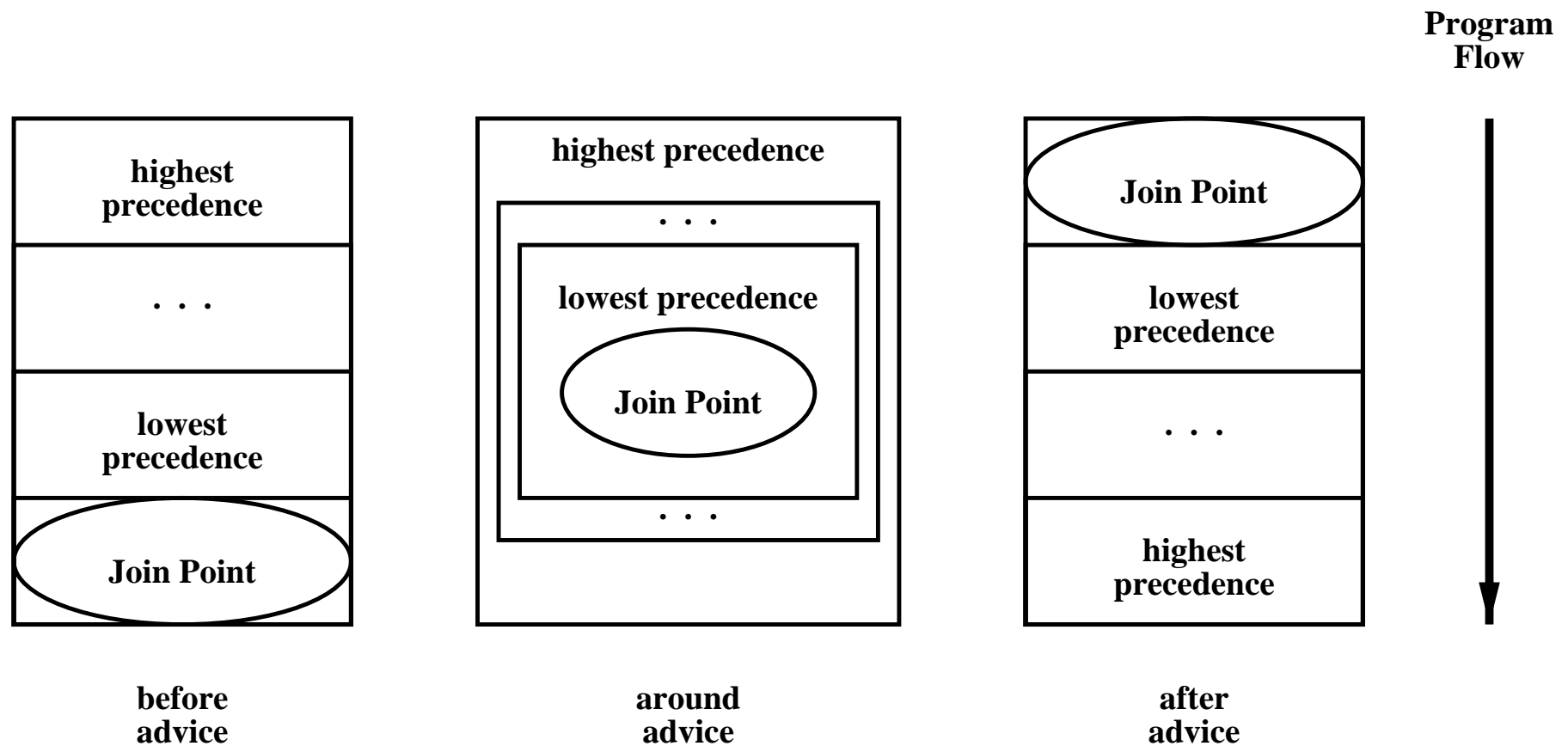
▣▣▣▣➡ can apply advice anywhere without handling `IOException`

# Aspect Precedence

---

What happens if several pieces of advice apply at the same join point?

▣▣▣▣➔ Assign precedence to aspects to control order of advice execution



# Aspect Precedence (cont.)

---

Syntax to declare aspect precedence:

```
declare precedence : TypePattern1, TypePattern2, ... ;
```

May occur in any aspect.

Says that anything matching type pattern 1 has higher precedence than anything matching type pattern 2, etc.

```
aspect CyclicPrecedence {  
    declare precedence : AspectA, AspectB;  
    declare precedence : AspectB, AspectA;  
}
```

OK iff aspects share no join points.

# Aspect Precedence (cont.)

---

If not declared, implicit rule for inheritance:

If `AspectA` extends `AspectB`, then `AspectA` has higher priority.

▣► possible to overrule advice from super-aspect.

If still not declared, implicit rule for advice within one aspect:

- If either are `after` advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

▣► first do something in the same order as they appear in the source

# Lexical Precedence Example

---

```
public aspect InterAdvicePrecedenceAspect {
    public pointcut jp() : call(* Main.perform());

    after() returning : jp() { System.out.println("<after1/>"); }

    before() : jp() { System.out.println("<before1/>"); }

    void around() : jp() {
        System.out.println("<around>");
        proceed();
        System.out.println("</around>");
    }

    before() : jp() { System.out.println("<before2/>"); }
}
```

# Lexical Precedence Example 2

---

```
aspect A {  
    before(): jp() {...1...}  
    after():   jp() {...2...}  
    before(): jp() {...3...}  
}
```

➡ 1 has higher precedence than 3

➡ 3 has higher precedence than 2

➡ 2 has higher precedence than 1

➡ leads to compiler error

# Abstract Aspects

---

Reminder: aspects can

- extend classes
- extend *abstract* aspects
- implement interfaces

Abstract aspects may contain

- abstract methods, like abstract classes
- abstract pointcut declarations

Abstract aspects are the key to writing *reusable aspects*.

# Abstract Aspects, Example

---

```
public abstract aspect AbstractTracing {  
    public abstract pointcut logPoints();
```

```
    before() : logPoints() {  
        System.err.println("Entering: " + thisJoinPoint);  
    }  
}
```

```
public aspect TraceMath extends AbstractTracing {  
    public pointcut logPoints() : call(* java.lang.Math.*(..)) ;  
}
```

```
public aspect TraceThreads extends AbstractTracing {  
    public pointcut logPoints() : call(* java.lang.Thread.start()) ;  
}
```

# Indented Tracing with Abstract Aspects

---

```
public abstract aspect IndentedTracing {
    protected int _ind = 0;

    protected abstract pointcut logPoints();

    before() : logPoints() { _ind++; }
    after()  : logPoints() { _ind--; }

    before() : call(* java.io.PrintStream.println(..)) &&
                within(IndentedTracing+) {
        for (int i = 0; i < _ind; i++) {
            System.out.print("  ");
        }
    }
}
```

# Indented Tracing (cont.)

---

```
public aspect TraceAspect2 extends IndentedTracing {  
    protected pointcut logPoints() : ... ;  
    before() : logPoints() {  
        // print out tracing stuff  
    }  
}
```

- ▣▣▣▣➔ precedence rules: extending aspect has higher priority
- ▣▣▣▣➔ tracing happens before incrementing `_ind`
- ▣▣▣▣➔ outermost level has indentation 0.

# Inner Aspects

---

Aspects can be (statically) 'inner', just like classes:

```
public class MyLinearSATSolver {  
    public boolean isSatisfiable(Clauseset c) {...}  
    ...  
    public static aspect TraceMe extends AbstractTracing {  
        public pointcut logPoints() :  
            call(* MyLinearSATSolver.*(..));  
    }  
}
```

static means: no access to instance fields of surrounding class.

# Aspect Instantiation

---

At runtime, aspects have fields like objects.

When do they get instantiated? Usually:

Instantiate an aspect once per program execution.

```
aspect Id {...}
```

```
aspect Id issingleton {...}
```

Implemented as singleton  static field in aspect class.

# Aspect Instantiation (cont.)

---

Per-object association:

```
aspect Id perthis(Pointcut) {...}
```

```
aspect Id pertarget(Pointcut) {...}
```

- create instance when the given pointcut becomes active
- one instance for every this/target object

Implemented using fields in the this/target objects picked out by the pointcut.

Typical use in abstract aspects

▣➡ different users don't need to share instance variables

# Per-object Association Example

---

Remember the Caching aspect:

Assume that the function to be cached is no longer static, but depends on the object on which it is called.

Idea: write reusable aspect

- with abstract pointcut for cached calls
- different cache for different objects using per-object association.

# Per-Object Example (cont.)

---

```
public abstract aspect CacheExpensiveCall pertarget(expCall()) {
    abstract pointcut expCall();

    private IntegerCache cache = new IntegerCache();

    int around(int x) : expCall() && args(x) {
        ...
    }
}

public aspect CacheFactorial extends CacheExpensiveCall {
    pointcut expCall() : call(int Numbers.func(int));
}
```

With execution(...) pointcuts, use perthis

# Aspect Instantiation (cont.)

---

Per-control-flow association:

```
aspect Id percfLOW(Pointcut) {...}
```

```
aspect Id percfLOWbelow(Pointcut) {...}
```

- create instance when the given pointcut, or the control flow below it, becomes active.
- a new instance at each entry.

Implementation: using a stack of active join points (?)

Application: e.g. create a new cache for recursive results at each top level call of binomical coefficient method.

# Member Introduction

---

Introducing methods to some class:

```
public aspect StringCountsWords {  
    public String String.countWords() {  
        ...  
    }  
}
```

usage:

```
String s = "some words";  
System.out.println(s + " has " + s.countWords() + " words.");
```

# Member Introduction (cont.)

---

Introducing fields:

```
private boolean Point.modified = false
```

Access modifier relative to defining aspect!

Possible to add fields to interfaces:

```
private boolean FigureElement.modified = false
```

Can even add default implementations to interface methods.

All this only to one type at a time, but wait a little...

# Default Implementation Example

---

```
public interface Nameable {  
    public void setName(String name);  
    public String getName();  
}
```

```
public class UsefulClass implements Nameable {  
    private String _name;  
    public void setName(String name) { _name = name; }  
    public String getName() { return _name; }  
  
    ... useful functionality  
}
```

▣▣▣▣➔ Repeat boilerplate for classes implementing Nameable

# Default Implementation Example (cont.)

---

```
public interface Nameable {
    public void setName(String name);
    public String getName();

    static aspect Impl {
        private String Nameable._name;

        public void Nameable.setName(String name) {
            _name = name;
        }

        public String Nameable.getName() {
            return _name;
        }
    }
}
```

# Member Introduction vs. per-Object Association

---

Per-Object association:

Keep required state (e.g. cache) in the aspect.

Alternative: keep it in the objects.

```
public aspect CachingAgain {
    private Cache Expensive.cache = new Cache();

    int around(int x, Expensive o) : call(int Expensive.f(int))
        && args(x) && target(o) {
        if (o.cache.contains(x)) {
            ...
        }
    }
}
```

▣▣▣▣▶ possible, but perthis/pertarget often cleaner.

# Modifying the Class Hierarchy

---

Add an `implements` clause to another class:

```
declare parents : FigureElement+ implements Cloneable;
```

▣▣▣▣➤ Default implementation can be added to interface!

Change superclass of another class:

```
declare parents : FigureElement+ extends MyAbstractSuperclass;
```

▣▣▣▣➤ very restricted, no multiple inheritance.

# Declaring Members in Many Types

---

You cannot say:

```
public void my.package.*.addListener(Listener l) {...}
```

to add a method in many classes.

But instead, you can say:

```
declare parents : my.package.* implements Observable;
```

and specify the default implementation by:

```
public Observable.addListener(Listener l) {...}
```

# Declaring Errors and Warnings

---

Exercise 4 was about access restrictions.

Often, these can be enforced at compile time

▣▣▣▣➔ declare new errors/warnings.

```
declare error: Pointcut : String;
```

```
declare warning: Pointcut : String;
```

▣▣▣▣➔ compiler complains at any point in the code, which will lead to a join point matching the pointcut.

Pointcut must be *statically determinable*:

May not use `cflow`, `cflowbelow`, `this`, `target`, `args`, `if`.

# Privileged Aspects

---

Usually, advice code has no access to private members of advised classes.

(Note that matching in pointcuts *does* see private members)

```
privileged public aspect A {  
    before(MyFriend f) : this(f) && ... {  
        System.out.println("My best friends secret: " +  
            f._privateField);  
    }  
}
```

Exercise: interaction with private member introduction.

# Conclusion

---

You should now know about:

- advice and exceptions
- exception softening
- abstract aspects
- aspect precedence
- aspect instantiation
- static cross-cutting facilities
- privileged aspects

Tomorrow you will learn about some usage patterns for aspects.