# Program specialization and metaprogramming

## *Laboratory work No. 9*

## Testing of AspectJ applications

### 1. Aims

To get acquainted with the advanced concepts of Aspect Oriented programming.

### 2. Tasks

Analyse and learn how to test aspect oriented programs.

### 3. Work

#### 1. Static Invariants

The easiest way to test with AspectJ is to use it to enforce static invariants, also known as compile time checks. These can be as simple as verifying a API is never called or as sophisticated as verifying your layered architecture is adhered to.

Create the first project now, Exercise1, using code downloaded from the course Moodle page.

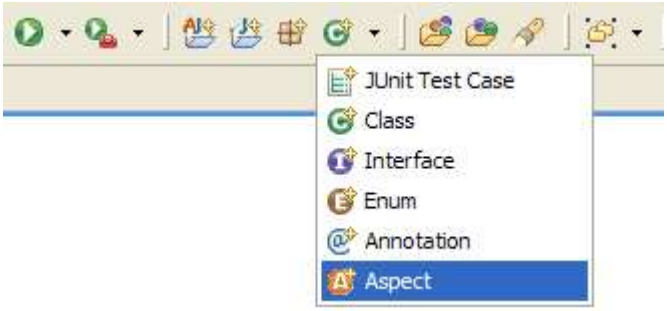
#### 1.a. Find old tracing

**Sample Exercise**: The main point of this exercise is to make sure your configuration works.

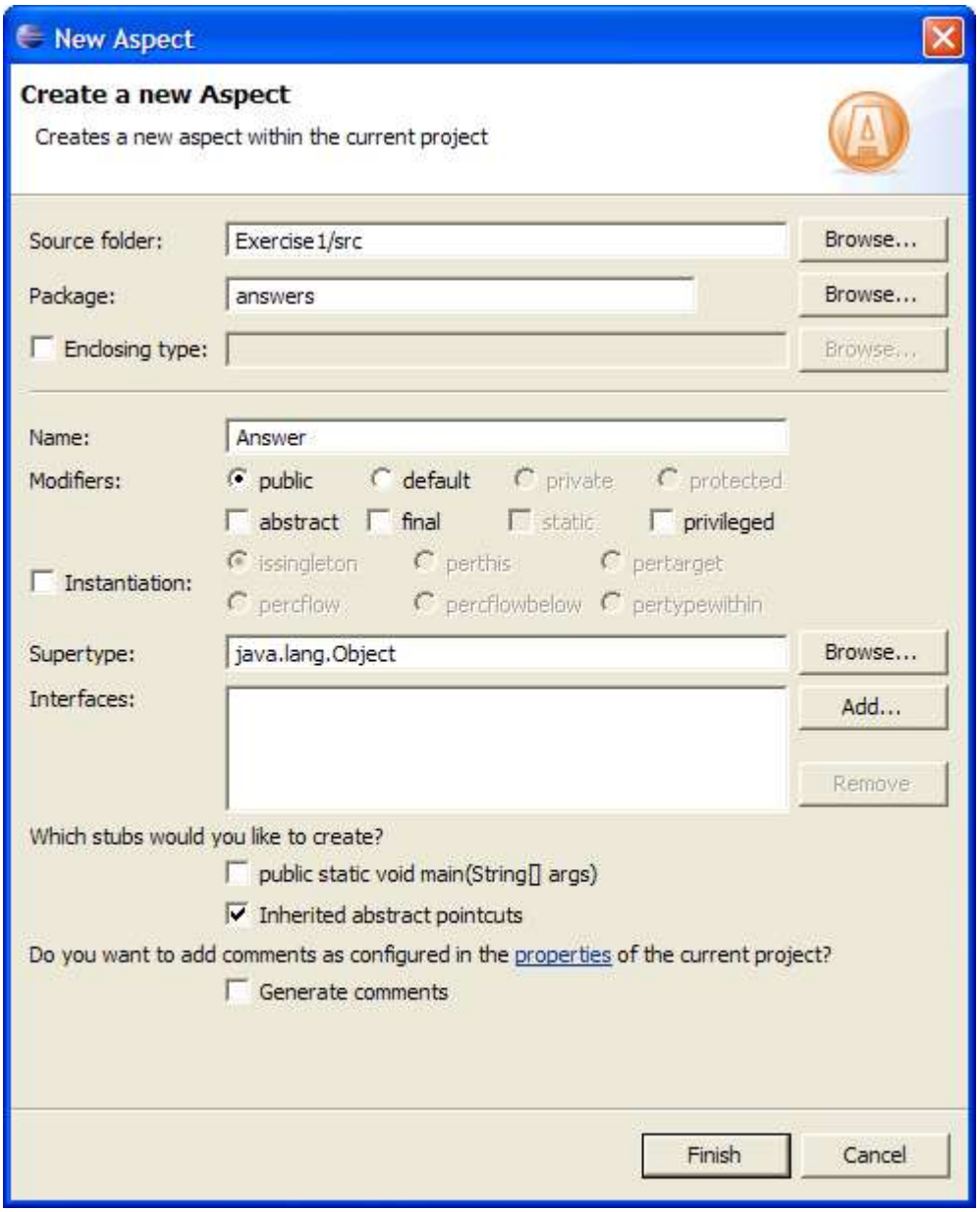
**Task:** Signal an error for calls to System.out.println.

The way that we are all taught to print "hello world" from Java is to use System.out.println(), so that is what we typically use for one-off debugging traces. It's a common mistake to leave these in your system far longer than is necessary. Type in the aspect below to signal an error at compile time if this mistake is made.

Create New aspect using New Java Class drop-down on the toolbar:



For this first aspect, fill it in as shown below:




After clicking Finish you should find yourself in the new aspect, change it to look like this following code

**Answer:**

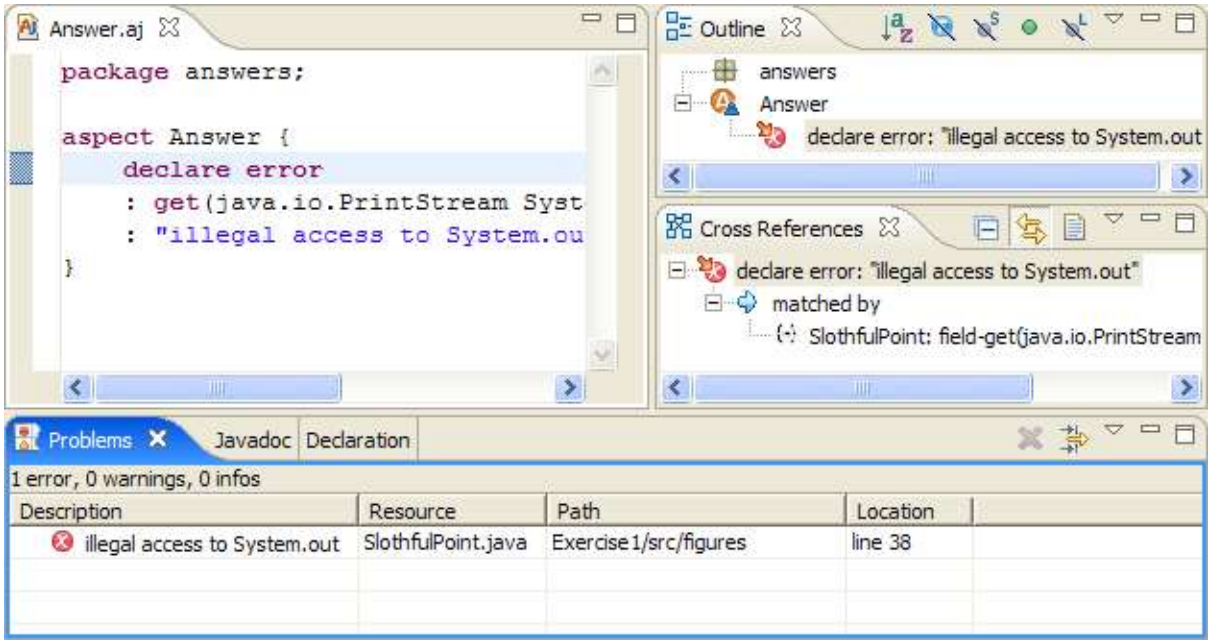
```
package answers;

import figures.*;

aspect Answer {
   declare error
      : get(java.io.PrintStream System.out) && within(figures..*)
      : "illegal access to System.out";
}
```

**Note:** For every task in this tutorial, you should work in the same Answer aspect that you create at the start of the exercise, comment out each previous answer as you move to the next task - if you don't then you may unexpectedly get some test failures.

After saving, a build will take place and you'll find one incorrect trace in SlothfulPoint:




Note that this answer does not say that the *call* to the println() method is incorrect, rather, that the field get of the out field is illegal. This will also catch those users who bind System.out to a static field to save typing. After successfully finding the error, edit your program to remove the illegal tracing call.

### 1.b. Mandate setters

**Task:** Signal a warning for assignments outside of setter methods.

**Tools:** set, withincode, the void set*(..) pattern

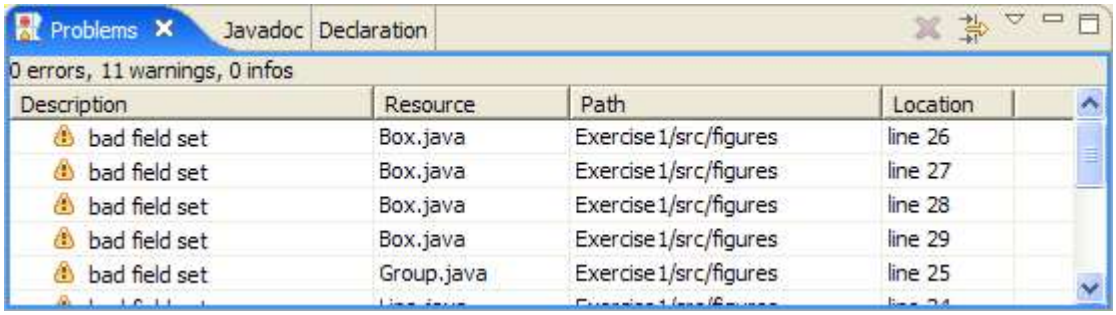
One common coding convention is that no private field should be assigned to outside of setter methods. Write an aspect to signal a warning at compile time for these illegal assignment expressions.

This is going to look like

```
aspect Answer {
   declare warning: <pointcut here> : "bad field set";
}
```

where the pointcut picks out join points of private field sets outside of setter methods. "Outside", here, means that the code for the assignment is outside the *text* of the setter.

Run the program. Make sure you get eleven warnings from this:




Wait to fix them until the next exercise.

### 1.c. Refine setters mandate

**Task:** Allow assignmnents inside of constructors.

**Tools:** the new(..) pattern

Look at some of the warnings from the previous exercise. Notice that a lot of them are from within constructors. The common coding convention is that no private field should be assigned to outside of setter methods *or constructors*. Modify your answer to signal an actual error at compile time (rather than just a warning) when such an illegal assignment expression exists.

After you specify your withincode  pointcut correctly, you'll still find that the convention is violated twice in the figures package. You should see the following two errors:

```
.\figures\Point.java:37 bad field set
.\figures\Point.java:38 bad field set
```

Rewrite these two occurrences so as not to violate the convention.

## 2. Dynamic invariants

The next step in AspectJ adoption is often to augment a test suite by including additional dynamic tests, sometimes called runtime checks.

### 2.a. Check a simple precondition

**Sample Exercise**: We've provided the answer to this exercise to get you started. Feel free to think a bit, but don't get stuck on this one.

**Task:** Pass tests.Test2a.

**Tools:** args, before

Write an aspect to throw an IllegalArgumentException whenever an attempt is made to set one of Point's int fields to a value that is less than zero.

**Answer:**

```
package answers;

import figures.*;

aspect Answer {
   before(int newValue): set(int Point.*) && args(newValue) {
      if (newValue < 0) {
         throw new IllegalArgumentException("too small");
      }
   }
}
```

### 2.b. Check another precondition

**Task:** Pass tests.Test2b.

**Tools:** call

Group is a FigureElement class that encapsulates groups of other figure elements. As such, only actual figure element objects should be added to Group objects. Write an aspect to throw an IllegalArgumentException whenever Group.add() is called with a null value.

### 2.c. Check yet another precondition

**Task:** Pass tests.Test2c.

**Tools:** target

Another constraint on a well-formed group is that it should not contain itself as a member (though it may contain other groups). Write an aspect to throw an IllegalArgumentException whenever an attempt is made to call Group.add() on a null value, or on the group itself.

Use a target pointcut to expose the Group object that is the target of the add call.

### 2.d. Assure input

**Task:** Pass tests.Test2d.

**Tools:** around advice

Instead of throwing an exception when one of Point's int fields is set to a negative value, write an aspect to trim the value to zero. You'll want to use around advice that exposes the new value of the field assignment with an args pointcut, and proceed with the trimmed value.

```
aspect Answer {
   void around(int val): <Pointcut> {
```

```
        <Do something with val>
        proceed(val);
    }
}
```

### 2.e. Check a postcondition

**Task:** Pass tests.Test2e

**Tools:** around advice

A postcondition of a Point's move operation is that the Point's coordinates should change. If a call to move didn't move a point by the desired offset, then the point is in an illegal state and so an IllegalStateException should be thrown.

Note that because we're dealing with how the coordinates change during move, we need some way of getting access to the coordinates both before *and* after the move, in one piece of advice.

### 2.f. Check another postcondition

**Task:** Pass tests.Test2f

**Tools:** the Rectangle(Rectangle) constructor, the Rectangle.translate(int, int) method.

FigureElement objects have a getBounds() method that returns a java.awt.Rectangle representing the bounds of the object. An important postcondition of the general move operation on a figure element is that the figure element's bounds rectangle should move by the same amount as the figure itself. Write an aspect to check for this postcondition -- throw an IllegalStateException if it is violated.

## 3. Tracing

Tracing is one of the classic AspectJ applications, and is often the first where AspectJ is used on deployed code.

### 3.a. Simple tracing

**Task:** Pass tests.Test3a.

**Tools:** Log.write(String), thisJoinPoint.toString(), execution, within

Write an aspect to log the execution of all public methods in the figures package. To do this, use the utility class Log (this is in the support package, so remember to import it into your answer aspect). Write a message into the log with the static write(String) method.

### 3.b. Exposing a value

**Task:** Pass tests.Test3b.

**Tools:** target

AspectJ can expose the target object at a join point for tracing. In this exercise, you will print not only the join point information, but also the target object, with the form

> *thisJoinPointInfo* at *targetObject*

### 3.c. More specialized logging

**Task:** Pass tests.Test3c.

**Tools:** args.

Write an aspect to log whenever a Point is added to a group. The args pointcut allows you to select join points based on the type of a parameter to a method call.

Look at the test case to see the trace message we expect you to write in the log.

### 3.d. Logging extended to checking an invariant

**Task:** Pass tests.Test3d.

**Tools:** inter-type field declaration

Make sure that a Point is never added to more than one Group. To do so, associate a boolean flag with each Point using an inter-type declaration, such as

```
boolean Point.hasBeenAdded = false;
```

Check and set this flag with the same kind of advice from your answer to problem (c). Throw an IllegalStateException if the point has already been added.

### 3.e. Better error messages for 3.d.

**Task:** Pass tests.Test3e.

Extend your solution to problem (d) by using the string representation of the Point's containing group as the msg part of the IllegalStateException.

## 4. Caching

Computation of the bounding box of Group objects needs to deal with all aggregate parts of the group, and this computation can be expensive. In this section, we will explore various ways of reducing this expense.

### 4.a. Make a constant override

**Task:** Pass tests.Test4a.

**Tools:** around, FigureElement.MAX_BOUNDS

Group's getBounds() method could be understood to be a conservative approximation of the bounding box of a group. If that is true, then it would be a legal (and much faster) implementation of getBounds() to simply always return a rectangle consisting of the entire canvas. The entire canvas is returned by the static field FigureElement.MAX_BOUNDS.

Write an aspect to implement this change. You can override Group's getBounds() method entirely with around advice intercepting the method.

### 4.b. Make a constant cache

**Task:** Pass tests.Test4b.

**Tools:** inter-type field.

Write an aspect instead that remembers the return value from the first time getBounds() has been called on a Group, and returns that first Rectangle for every subsequent call.

*Hint: You can use an inter-type declaration to keep some state for every Group object.*

### 4.c. Invalidate

**Task:** Pass tests.Test4c.

**Tools:** before

While caching in this way does save computation, it will lead to incorrect bounding boxes if a Group is ever moved. Change your aspect so that it invalidates the cache whenever the move() method of Group is called.

## 4. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?