Fall, 2005 Name: _____

Com S 541 — Programming Languages 1

Test on Aspect-Oriented Languages and AspectJ

This test has 7 questions and pages numbered 1 through 7.

Special Directions for this Test

This test is open book and notes.
If you need more space, use the back of a page. Note when you do that on the front.
This exam is timed. We will not grade your exam if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire exam so that you can budget your time.
Clarity is important; if your diagrams or programs are sloppy and hard to read, you will lose points.
Correct syntax also makes a difference for programming questions.
When you write AspectJ code on this test, you may use anything in the Java standard libraries, or in the packages `org.aspectj.lang` or `org.aspectj.lang.reflect` without writing it in your test. You are encouraged to define aspects, classes, interfaces, fields, and methods not specifically asked for if they are useful to your programming; if they are not in the standard libraries or the packages named above, please write them into your test.

For Grading

Problem	Points	Score
1(a)	5	
1(b)	10	
2	10	
3(a)	15	
3(b)	10	
4	20	
5	10	
6	10	
7	10	
total	100	

1. This is a problem about using advice to change the way "NaN" is treated as the first argument of a method execution.

   NaN, the "not a number" double-precision floating point in Java, `Double.NaN`, is used to represent undefinedness; for example the result of `0.0/0.0` is `Double.NaN`. Since `Double.NaN` behaves strangely in comparisons (e.g., `Double.NaN == Double.NaN` is false!), you should use the static method `Double.isNaN`, which takes a `double` argument and returns a `boolean`, to test whether a `double` is `Double.NaN`.

   You are to write an aspect `floats.NoNaNArgs` (i.e., `NoNaNArgs` in the package `floats`) containing:

   (a) (5 points) A public pointcut, `isNaNExecs`, that describes executions of the static method `Double.isNaN`.

   (b) (10 points) Advice that throws an `IllegalArgumentException` if the first argument of any method execution, other than a call to `Double.isNaN`, is `Double.NaN`. (If the first argument is not a double, or is not `Double.NaN`, then the method execution is unaffected by this advice. Note that `IllegalArgumentException` is a subtype of `RuntimeException`.)

   An example test case, which succeeds given the advice you are to write, is shown on the next page.

The following is a testcase for the problem on the previous page.

```
package tests;
import junit.framework.TestCase;
/** Test of NoNaNArgs aspect. */
public class NoNaNArgsTest extends TestCase {
    /** Standard testcase constructor */
    public NoNaNArgsTest(String name) { super(name); }
    /** Run the tests */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(NoNaNArgsTest.class);
    }

    static void foo(double y) { }
    static boolean foo2(double y, double z) { return y == z; }

    /** With advice, this succeeds by catching the exception. */
    public void testCallOneArg() {
        try {
            foo(Double.NaN);
        } catch(IllegalArgumentException e) {
            return;
        }
        fail("Running foo(Double.NaN) didn't throw exception");
    }

    /** With advice, this succeeds by catching the exception. */
    public void testCallTwoArg1() {
        try {
            foo2(Double.NaN, 4.0);
        } catch(IllegalArgumentException e) {
            return;
        }
        fail("Running foo2(Double.NaN, 4.0) didn't throw exception");
    }

    /** With advice, this checks that no exception is thrown
     *  when an argument other than the first is NaN. */
    public void testCallTwoArg2() {
        foo2(3.0, Double.NaN);
    }

    /** Calls to Double.isNaN shouldn't be affected. */
    public void testCallToIsNaN() {
        Double.isNaN(Double.NaN);
    }
}
```

2. (10 points) Scala also uses `Double.NaN` for the same purpose as in Java.
   How hard would it be to solve the previous problem in Scala? Explain briefly.

3. Consider the following interface.

```
package floats;
/** A marker interface for code in which NaN is an OK result. */
public interface NaNResultsOk { }
```

   In AspectJ, without changing any existing code, write an aspect `NaNResultCheck` such that:

   (a) (15 points) It advises all executions of all methods in all types that do *not* implement `NaNResultsOk` and whose return type is `double`, so that if they attempt to return `Double.NaN` the advice causes the execution to instead throw a `RuntimeException`. (If the result is not a double, or if the method is in a subtype of `NaNResultsOk`, or if the result of the execution is not `Double.NaN`, then the method execution is unaffected by this advice.)

   (b) (10 points) It makes the classes `Math` and `StrictMath`, in the package `java.lang`, implement the `NaNResultsOk` interface. (You should assume that AspectJ can weave into these classes.)

   An example test case, which succeeds given the advice you are to write, is shown on the next page.

The following is a testcase for the problem on the previous page.

```
package tests;
import junit.framework.TestCase;
/** Test of NaNResultCheck aspect. */
public class NaNResultCheckTest extends TestCase {
    /** Standard testcase constructor */
    public NaNResultCheckTest(String name) { super(name); }
    /** Run the tests */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(NaNResultCheckTest.class);
    }

    class CheckMe {
        public double bad() { return Double.NaN; }
        public double maybe(double x) { return x/x; }
        public double good() { return 0.0; }
    }

    class ImOK implements floats.NaNResultsOk {
        public double okay(int x) { return Double.NaN; }
        public double noprob() { return 0.0; }
    }

    /** With advice, this succeeds by catching the exception. */
    public void testCheckMeBad() {
        try {
            new CheckMe().bad();
        } catch(RuntimeException e) {
            return;
        }
        fail("Running new checkMe.bad() didn't throw exception");
    }

    /** With advice, this succeeds by catching the exception. */
    public void testCheckMeMaybe() {
        try {
            new CheckMe().maybe(0.0);
        } catch(RuntimeException e) {
            return;
        }
        fail("Running new CheckMe().maybe(0.0) didn't throw exception");
    }

    /** Test that allowed calls don't throw an exception. */
    public void testAllowedCalls() {
        new CheckMe().maybe(3.0); // returns 1.0
        new CheckMe().good();   // returns 0.0
        new ImOK().okay(541);  // NaN, but in subtype of NaNResultsOk
        new ImOK().noprob();  // returns 0.0
        Math.acos(75.0);      // NaN, but in subtype of NaNResultsOk
        StrictMath.log(-75.0); // NaN, but in subtype of NaNResultsOk
    }
}
```

4. (20 points) Java's `compareTo` method is defined in the interface `Comparable` with the following signature:

```
int compareTo(Object o);
```

According to the documentation, this method Returns a negative int, zero, or a positive int if `this` object (i.e., the receiver) is less than, equal to, or greater than the specified object (i.e., the argument, `o`), respectively.

In AspectJ, write an aspect `StandardizeComparable` in a package `stds` to make sure that, whenever a `compareTo` method call to a subtype of `Comparable` returns, it returns -1, 0, or +1, when the original call would have returned a negative int, 0, or a positive int, respectively.

For example, with the required aspect, if a `compareTo` method call would have returned -27, it would instead return -1. Similarly, a call of `compareTo` that would have returned 541 instead would return 1.

5. (10 points) Would a problem like the previous one be difficult to solve in Scala? Explain briefly.

6. (10 points) Suppose you had to write a very clear program to describe the mathematical semantics of a small programming language, say for a paper to be published in a conference. Which language, out of Haskell, Scala, or AspectJ would you pick for this task. State the language you pick (one only!) and explain your choice briefly.

7. (10 points) Briefly compare the advantages of AspectJ over Scala for program maintenance.