Fall, 2002 Name: ______________________

Com S 541 — Programming Languages 1

Test on Aspect-Oriented Programming Languages and AspectJ

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 5.
This test is open book and notes. However, you are not to share books and notes with other students.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write AspectJ code on this test, you may use anything in the Java libraries without writing it in your test; however, you should use an appropriate `import` when you do that. You are encouraged to define methods and classes not specifically asked for if they are useful to your programming; if they are not in Java, write them into your test.

Base Program

In the first two problems we will use the following code as a base program.

```
public class AddIntArray {
    private int [] vals;

    /** Initialize this class */
    public AddIntArray(int [] a) {
        this.vals = (int [])a.clone();
    }

    /** return the size of the array. */
    public int size() {
        return vals.length;
    }

    /** Add the values in the array from indexes from to end inclusive. */
    public int getSum(int from, int end) {
        if (from > end) {
            return 0;
        } else {
            return vals[from] + this.getSum(from+1, end);
        }
    }

    /** Return a string representing this object. */
    public String toString() {
        String ret = "[";
        for (int i = 0; i < vals.length-1; i++) { ret += vals[i] + ", "; }
        if (vals.length > 0) { ret += vals[vals.length-1]; }
        return ret + "]";
    }
}
```

1. (20 points) In AspectJ, without changing `AddIntArray` (see the previous page), write an aspect `GetSumChecking` that checks the validity of calls to `AddIntArray`'s `getSum` method. That is, the aspect should define advice that checks that when `getSum` is called, `from` is non-negative and less than or equal to `end + 1` and that `end` is strictly less than the size of the array in the target object. If this condition is violated, the code should throw an `AssertionError` exception whose string message includes the line number and file name where an invalid call was made. For example:

```
public class TestGetSumViolation {
    public static void main(String [] argv) {
        AddIntArray a = new AddIntArray(new int[] {5, 4, 1});
        int sum02 = a.getSum(0, 2); int sum12 = a.getSum(1, 2); // ok
        int sum22 = a.getSum(2, 2); int sum32 = a.getSum(3, 2); // ok
        int sum01 = a.getSum(0, 1); int sum11 = a.getSum(1, 1); // ok
        int sum21 = a.getSum(2, 1); // ok
        int sum31 = a.getSum(3, 1); // invalid  // line 8
        int sum42 = a.getSum(4, 2); // invalid
        int sum_12 = a.getSum(-1, 2); // invalid
        int sum03 = a.getSum(0, 3); // invalid
        System.out.println("sum02 == " + sum02);
    }
}
```

would produce the following output.

```
java.lang.AssertionError: precondition of getSum violated by call
    on line 8 of file TestGetSumViolation.java
        at GetSumChecking.before0$ajc(GetSumChecking.java:15)
        at TestGetSumViolation.getSum$method_call6(TestGetSumViolation.java:11)
        at TestGetSumViolation.main(TestGetSumViolation.java:11)
Exception in thread "main"
```

Your task is to write an aspect that would produce output like the above in general. Note that the first two lines of the output above are produced by the aspect you are to write (and not by the exception mechanism).

2. (20 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `GetSumTiming` that prints out the number of milliseconds taken by each top-level (outermost) call to `AddIntArray`'s `getSum` method. (Hint: use Java's `System.currentTimeMillis()` method, which returns a `long` integer.) For example, with the program `TestGetSumViolation` shown in the previous problem, the program's output would be like the following.

```
Call to [5, 4, 1].getSum(0, 2) took: 16 ms
Call to [5, 4, 1].getSum(1, 2) took: 5 ms
Call to [5, 4, 1].getSum(2, 2) took: 0 ms
Call to [5, 4, 1].getSum(3, 2) took: 2 ms
Call to [5, 4, 1].getSum(0, 1) took: 0 ms
Call to [5, 4, 1].getSum(1, 1) took: 0 ms
Call to [5, 4, 1].getSum(2, 1) took: 0 ms
java.lang.AssertionError: precondition of getSum violated by call
    on line 8 of file TestGetSumViolation.java
        at GetSumChecking.before0$ajc(GetSumChecking.java:15)
        ... // details of the exception suppressed
```

3. (10 points) Write, in AspectJ, a pointcut declaration that describes either read or write access to any protected field. For example, if class `B` declares `protected int x;`, then the set of joinpoints denoted by the pointcut should include the read and write accesses of `x`. Of course, what you are to write should handle all possible variables of all possible types.

4. (15 points) Is it possible to write, in AspectJ, a pointcut declaration that describes read or write access to a protected field from code that is not in a subclass but is within the same package as the declaration of the field being accessed? For example, if class `B` in package P declares `protected int x;` and class `C`, also in package P reads or writes `x`, but `C` is not a subclass (or descendent) of `B`, then the set of joinpoints denoted by this pointcut include the read and write accesses in `C` of `x`. Of course, what you are to write should handle all such variables in all possible packages.

   If you can do so, write the pointcut declaration, if not, describe why you cannot.

5. (15 points) What is the difference between AspectJ's `after`, `after returning`, and `after throwing` advice? Explain how they differ, and the advantage of having all three kinds of advice in AspectJ.

6. (20 points) Is it useful to have a distinction between `call` and `execute` join points in an aspect-oriented extension to Smalltalk? Explain the advantages and disadvantages in terms of your design for Aspect Smalltalk.