AspectJ is a pretty powerful language intended for building *aspects* for Java. Aspects are constructs that help modularize *cross-cutting concerns* in object-oriented software. In this handout, we will study some of the features of AspectJ, and learn how to use these features.

1 Getting Started

The best (easiest) way to program using AspectJ is to use the Eclipse IDE for Java. There is a plugin for Eclipse called AJDT (AspectJ Development Tools) that allows for aspect development in Eclipse with full compiler support. Moreover, the AJDT includes some pretty nifty tools such as the Aspect Visualizer that make it a pleasure to program!

1.1 Setting up ADJT in Eclipse

I am assuming that you already have eclipse installed on your computer. If not, please go to [http://www.eclipse.org](http://www.eclipse.org) and get the latest version of the IDE.

Next, download the AJDT plugin for Eclipse at [http://www.eclipse.org/downloads/index.php](http://www.eclipse.org/downloads/index.php). Follow the instructions on that page to set up AJDT on your computer. Once you have installed the plugin, just restart Eclipse, and you should be all set to start programming with AspectJ.

1.2 Using Command-line Tools

If you don’t want to use Eclipse, you can use the AspectJ command-line tools to do your development as well. Download the aspectJ compiler at [http://www.aspectj.org](http://www.aspectj.org). You should find instructions on that website on how to set up your environment and use the tools.

2 Defining Pointcuts

A *join point* is an identifiable point in the execution of a program, and a *pointcut* is a program construct that selects join points and collects context at those points. Most of the work involved in aspect-oriented programming lies in identifying the join points where a particular aspect is to be weaved in. AspectJ pointcuts allow us to identify a variety of different kinds of join points such as method call (and return), class initialization, object instantiation, etc. In this section, we will see a few examples of what kind of pointcuts we can define using AspectJ.

*Download the LoggingSamples source archive from the course website follow along. The notes below make references to the code in this archive.*
2.1 Method Call

Consider the `StackImpl` class, and the class `StackTester` that uses `StackImpl`. Suppose that we want to log all method calls to the `StackImpl` object, and moreover, log the values of method parameters and return objects. How would we do this using an aspect? What we want to do is to insert some logging code every time a method on any `StackImpl` object is invoked, and similarly some logging code on the way back from the method to the client.

AspectJ offers us a choice of two kinds of pointcuts to choose from in this case: `call` and `execution`. For now, let’s just treat these two as the same; we will see the difference between the two in a little bit. Look at the sequence diagram below; the join points captured by `call` and `execution` pointcuts is shown.

As you can see from the picture, at least for now, there is no real difference as to whether we use a `call` pointcut or an `execution` pointcut to capture all method invocations going to all `StackImpl` objects. So here, let’s use the `execution` pointcut:

```
public pointcut publicStackMethods():
    execution (public * StackImpl.*(..));
```

This pointcut captures all the join points in the `execution` of every `public` method in the `StackImpl` class. The pointcut does not care about the name of the method, the number and types of parameters, or the return type.

If you look at the source code for the `StackImpl` class in Eclipse, you should see the little “pointcut arrows” at the start and finish of every `public` method in the class. Now look at the `StackTester` class. There are no pointcut markers. This shows us that the join points captured is only after the method invocation has reached the target object.

Now, let’s find out the real difference between the `call` and `execution` pointcuts. Change the `publicStackMethods` pointcut to be an `execution` pointcut:

```
public pointcut publicStackMethods():
    call (public * StackImpl.*(..));
```
Now go back and look at both classes (StackImpl and StackTester). Do you see any pointcut markers at all? NO! Where did they go? In what way should you modify the pointcut in order to capture the calls in StackTester?

```
public pointcut publicStackMethods():
    call _____________________________ ;
```

Apart from the difference shown above, there is another important difference between call and execution pointcuts. And this is in the kinds of context information that you can access from these pointcuts. AspectJ provides two keywords — this and target to refer to the caller object and callee object in a call pointcut. However, in an execution pointcut, the keyword this refers to the target object. But there is no way to reach back to the caller. So if you need any information from (or about) the caller object, you should use the call pointcut.

2.2 Wildcards in Pointcut Definitions

We wrote a pointcut above to capture all join points during the execution of public methods in StackImpl. What if we wanted to capture all join points during the execution of all objects? We can use a wildcard to signify the name of the class:

```
public pointcut publicMethods():
    execution (public * *.*(..));
```

This pointcut now says, “give me all the join points in any public method in any class, regardless of argument list or return type”. This means that it will also capture all the join points inside the Logger class as well. We already know what Logger does, so we don’t want to log those methods. So let’s modify the pointcut to exclude all the join points in Logger:

```
public pointcut publicMethods():
    execution (public * *.*(..)) && !within(Logger);
```

Now this pointcut captures all the join points in StackImpl and in StackTester, but not in Logger; this is what we wanted to do. A quick check in the source code of the two classes shows us that this is true: there are aspect arrows at every public method in StackImpl and StackTester.

Let’s try one more experiment: All classes in Java extend Object. So we should be able to modify the pointcut to capture public methods in all classes (excluding Logger) to read as follows:

```
public pointcut publicMethods():
    execution (public * Object.*(..)) && !within(Logger);
```

The theory here is that every class in Java has Object in its class hierarchy, meaning that every instance of any class in Java “is an” Object. Now go back and look for aspect arrows in the source code of StackTester: there are none! In StackImpl, there is a marker only in one method — toString(). Why is this?

How would you modify the pointcut (without reverting back to * instead of Object) to go back to the state of capturing all the public methods in StackImpl and StackTester?

```
public pointcut publicMethods():
    execution (______________________) && !within(Logger);
```
3 Advice

Pointcuts allow aspects to identify specific join points in classes. What do they do at these join points? This is specified using advices. AspectJ allows the insertion of advice code before, after, and around any pointcut. Refer to the AspectJ notes handed out previously for examples of these different kinds of advice. The key thing to remember is that a before or after advice cannot stop method from executing, whereas an around advice can.

4 Aspect Precedence

When a project consists of several aspects, and moreover, multiple aspects define advice at a single pointcut, it becomes necessary to consider the semantics of aspect precedence. Consider the Home Security example in the HomeSecurity source archive. The example consists of two classes — Home and TestHome, and two aspects — HomeSecurityAspect and SaveLightsAspect. Both aspects define the same two pointcuts:

```java
call (void Home.enter())
call (void Home.exit())
```

How can we predict which of the two aspects will be applied first? The short answer is that as things stand, we cannot. The order in which aspects are applied are completely non-deterministic. The aspect weaver picks any order at the time of weaving. Fortunately however, there is a way of forcing a particular ordering in which aspects should be applied. The construct that AspectJ provides for this is the declare precedence statement. Writing the following establishes that the HomeSecurityAspect has higher precedence than the SaveLightsAspect, and will therefore be applied at the highest level:

```java
declare precedence: HomeSecurityAspect, SaveLightsAspect;
```

A point to note here is what this precedence means in the context of before and after advice. In the case of before advice, the aspects are applied in the order of highest to lowest precedence, and in the case of after advice, they are applied in the order of lowest to highest precedence.

4.1 Intra-Aspect Advice Precedence

While there is a syntactic construct for explicitly specifying the precedence of aspects over one another, there is no such construct for specifying the order in which advices inside a single aspect should be applied. Look at InterAdvicePrecedenceAspect in the IAP source archive. The only way to specify ordering of advice is the order in which they appear in the aspect source code. Play with this aspect a little to see how different combinations produce different output.

5 Static Cross-Cutting

In addition to all the different kinds of pointcuts used to specify join points during the execution of a program, AspectJ also allows the static extension of a class. As a companion to this section, look at the Fibonacci folder in the Fibonacci source archive. The class Fibonacci computes
the Fibonacci series. The parameter to `fibonacci()` specifies how many numbers in the series it should compute. The program as is currently implemented is implemented using recursion. The execution time for the `fibonacci()` method is exponential, meaning that to compute the value of the 100th number in the series will take $O(2^{100})$ time.

`FibonacciSaverAspect` is designed to change this implementation (without modifying the original class) to use a dynamic programming solution. Dynamic programming is a technique used in algorithm design. At its core, the technique uses the concept of caching values that will be needed in the future. In our example, in order to compute `fibonacci(n)` for some `n`, we also need the values of `fibonacci(n-1)` and `fibonacci(n-2)`. And since we need to compute these values recursively, we end up having to use all values of `fibonacci(m)`, where $1 \leq m < n$. The regular solution to the problem (which is presented in the `Fibonacci` class) is to compute each of these values over and over again, resulting in exponential execution time.

The dynamic programming approach to computing the fibonacci series will only compute the value of `fibonacci(m)` for any `m` exactly once. As soon as the value is computed the first time, it is stored for future reference. From the next time on, the value is simply looked up from this table, and not computed. This saves a lot of time, resulting in the program actually terminating.

In order to do this, the `FibonacciSaverAspect` extends the `Fibonacci` class statically to add a private data member called `fibValues`, an array of `long`, that stores the computed fibonacci values. The aspect also includes an around advice to intercept any call to `fibonacci()`. The advice checks the `fibValues` array to check if the requested fibonacci value has already been computed, and if so, the method invocation is never sent to the target; the value from the array is returned.

6 Aspect Association

For this section, download the `AssociationSamples` zip archive from the course website, and look up examples.

When an aspect is included in a Java application, how many instances of the aspect are created? When are they created?

By default, only one instance of every aspect is created in a AspectJ/Java application. This same instance is applied to every part of the application that the pointcuts define. At any point, if any part of the application (either a class or a different aspect) want to get a hold of the aspect instance, they can use the `aspectOf()` static method that is present in every AspectJ aspect. For instance, the following statement acquires the instance of the `AssociationDemoAspect`:

```java
AssociationDemoAspect ada = AssociationDemoAspect.aspectOf();
```

Although the default case is to just create a single instance of an aspect, we can force the creation of more instances. When multiple aspect instances are requested, the request is made by including a pointcut in signature of the aspect, specifying which object the aspect must be associated with:

```java
public aspect AssociationDemoAspect perthis(this(MyClass))
{
    ...
}
```

creates a new instance of this aspect for every instance of `MyClass`. Experiment with this example by using different combinations of `perthis`, `pertarget`, `percflow`, `percflowbelow`. 