AspectJ Tutorial

• Introduction to AspectJ
  ▫ Aspect-oriented paradigm
  ▫ AspectJ constructs
• Types of Join Points
  ▫ Primitive
  ▫ Lexical designators
  ▫ Type designators
  ▫ Control flow
• Types of Advice
  ▫ Before
  ▫ After
  ▫ Around
• Receptions Join Points
  ▫ Java method dispatch
  ▫ Vs. executions join points
• Exposing Context with
  ▫ thisJoinPoint, thisStaticJoinPoint
  ▫ Other ways to expose context modularly
• Aspect Instantiation
  ▫ The of clause
• Lexical Introduction
  ▫ Additional members
  ▫ Extension & implementation
  ▫ Private introduction
• Aspect Extension
  ▫ Abstract aspects
• Aspect Privilege
• Aspect Composition
  ▫ Priority & domination
  ▫ Recursion
• Additional Topics
  ▫ Throwing checked exceptions
  ▫ Current compiler limitations
AspectJ & AOP

• General-purpose aspect-oriented extension to Java
• Grew out of coordination library (COOL)
• Aspect-oriented programming allows you to modularize concerns that would otherwise cut across object-oriented program logic
  □ Logging/Tracing
  □ Session Management
  □ Coordination
• Why use aspects? MODULARITY
  □ Conditional compilation made easy
    ◦ Implementation of pluggable features
      ▪ debugging
  □ Aspects can implement features necessary for correctness of programs
    ◦ synchronization
    ◦ reactivity
  □ Aspects can introduce space or time optimizations
    ◦ caching
So, What is an Aspect?

- Modular unit of crosscutting implementation

- An AspectJ *aspect* is a crosscutting *type* consisting of
  - *advice* on *pointcuts*
  - *lexical introduction* of behavior into other types

- Like classes, aspects can have internal state and behavior, can extend other aspects and classes, and can implement interfaces
Advice on Join Points

- **Join point**: a well-defined location at a point in the execution of a program
  - the execution of the method `public void A.foo(int)`
  - the static initialization of class `A`
- **Pointcut**: a set of join points
  - all method calls to class `B` within class `A`
  - all mutations of fields of class `A` outside of `A`'s subclasses
- **Advice**: code designed to run automatically at all join points in a particular pointcut
  - can be marked as `before`, `after`, or `around` (in place of) the join points in the pointcut
- **Lexical introduction**: adding functionality to a class `in place` (as opposed to `extending` it)
  - For example, making class `A` implement `Cloneable`
Composition of Join Points

- Use \( \&\& \mid \mid \mid \! \)
- Use defined pointcuts in other pointcuts

pointcut fooCalls():
    calls(int Bar.foo()) \&\& within(MyClass);

pointcut interestingClasses():
    instanceof(MyPackage..*);

pointcut interestingReceptions():
    ( receptions(* *(..)) | receptions(new(..)) )
    \&\& interestingClasses();

pointcut nonstaticMethods():
    executions(!static *(..));
Types of Join Points

Primitive:

initializations(\textit{GTN})
staticinitializations(\textit{GTN})
receptions(\textit{Signature})
eexecutions(\textit{Signature})
calls(\textit{Signature})
callsto(\textit{PCD})

sets(\textit{Signature})[\textit{oldVal}][\textit{newVal}]
gets(\textit{Signature})[\textit{value}]

handlers(\textit{throwable type name})

Lexical extents:

within(\textit{GTN})
withinall(\textit{GTN})
withincode(\textit{Signature})

Type designators:

instanceof(\textit{GTN})
hasaspect(\textit{GAN})

Control Flow:

cflow(\textit{PCD})
cflowtop(\textit{PCD})
Types of Advice

before() : pointcut { advice }

after() returning() : pointcut { advice }

- advice runs if join point computation concludes successfully

after() throwing() : pointcut { advice }

- advice runs if join point computation throws an exception

after() : pointcut { advice }

- advice runs in either case, and after the others

around() returns type : pointcut { advice }

- return type widening
- advice must return a value

- advice must explicitly act to proceed with join point computation if the computation is to continue at all

- Because the flow of control dips through the advice, it can modify method arguments and/or the return value

- Implements a middle wrapping layer that is completely modular -- neither caller or receiver need to know about it
Advice Priority

• If more than one advice block affects the same join point, they operate in this order:
  - *around advice* is run *most specific first*
  - *before advice* is run *most specific first*
  - *after advice* is run *least specific first*

• Of course, if any *around advice* executes that does not continue with join point computation, no other advice runs for the join point
Receptions Join Points

• Related to the idea of object-oriented message-passing
• Java method dispatch
  ▫ There are two ways to execute public, non-static methods in Java:
    ◦ `a.foo()` - dispatch occurs at runtime based on runtime type of `a`
    ◦ `super.foo()` - the implementation to use is known at compile time
• `receptions` join points occur at runtime dispatch
  ▫ A `receptions` join point never catches superclass calls
  ▫ A `receptions` join point does not occur at a place in the code - cannot be used with lexical constructs like `within`!
• `receptions` vs. executions join points
• `receptions` vs. calls join points
Exposing Context - Part I

• `thisJoinPoint` is statically typed as `JoinPoint` but is actually a `MethodExecutionJoinPoint`, a `HandlerJoinPoint`, or whatever `JoinPoint` is actually an interface hierarchy
  □ Cast `thisJoinPoint` to the proper type (if necessary for the information you need)
• `thisStaticJoinPoint`
  □ a lightweight join point object
  □ similar to `thisJoinPoint` but only static information is available
    □ a `StaticJoinPointException` is thrown if you ask for more
• Package `org.aspectj.lang` contains:
  □ `JoinPoint`
  □ `Signature`
  □ `SourceLocation`
• Package `org.aspectj.lang.reflect` contains:
  □ `JoinPoint` subinterfaces
  □ `Signature` subinterfaces
  □ `StaticJoinPointException`
• These packages are *not* automatically imported for you
Exposing Context - Part II

• If we have a pointcut:

    pointcut fooCalls() : calls(Bar.foo(int));

... but we really want to know what that int is, we can write:

    pointcut fooCalls(int i) : calls(int Bar.foo(i));

• We then write advice constructs like these:

    before(int i) : fooCalls(i) {
        System.out.println("The int is " + i + "!");
    }

    after(int i) returning(int j) : calls(int Bar.foo(i)) {
        System.out.println("Bar.foo(" + i + ") returned " + j);
    }

    around(int i) returns int : receptions(int Bar.foo(i)) {
        // double the argument, halve the result
        return proceed(2*i)/2;
    }
Exposing Context - Part III

• Exposing context can be very useful

```java
pointcut guardedInts(int oldval, int val) :
    sets(int Foo.*)[oldval][val];
around(int oldval, int val) returns int :
    guardedInts(oldval, val) {
        if(Math.abs(oldval - val) > 5)
            throw new RuntimeException("Delta too big -> " +
                oldval + " to " + val);
        return proceed(oldval, val);
    }
```
Aspect Instances

- Aspects cannot be instantiated with new and may only have nullary constructors, even if they extend classes
- of clauses
  - of eachJVM()
    - This is the default, one aspect instance for the whole virtual machine
    - You can use `FooAspect.aspectOf()` to get the singleton instance of `FooAspect`
  - of eachobject( PCD )
    - Associate a `shadow` aspect instance with every object in the `PCD`
    - Each pointcut has an implicit `hasAspect()`
    - You can use `FooAspect.aspectOf(obj)` to get the instance of `FooAspect` associated with `obj`
    - throws an `NoAspectBoundException` on error
  - of eachclass( PCD )
    - Part of the AspectJ language, but not yet implemented in the compiler
  - of eachcflowroot( PCD )
    - Control flow entering each join point in the `PCD` get an aspect instance
Lexical Introduction

• Making a class extend another or implement an interface
  
  ```java
  Foo +extends Bar;
  Foo +implements Cloneable;
  ```

• Introduction of state and behavior
  
  ```java
  protected static int Foo.i;
  public Vector (Foo || Bar).aVector = new Vector();
  ```

• Or, if you have a lot of classes to introduce into...
  
  ```java
  interface I {}
  String I.foo() { return "some string"; }
  int I.someInt = 5;
  (Foo || Bar || Bat || Bam || SomePackage..*) +implements I;
  ```

• Private introduction
  
  □ Private to the aspect, not to the class
  □ Guaranteed not to cause conflicts
    ○ Currently a problem with making classes Serializable since private
      writeObject() and readObject() methods are required
Aspect Extension

- Aspects can extend classes other aspects that are explicitly labelled abstract

- Pointcuts are inherited

- Abstract pointcuts can be extended

- `of` clause inherited
Aspect Privilege

• Way too powerful right now, may be more controlled later
  - Declare an aspect privileged and it has access to all private members of all classes

```java
class C {
    private int i;
    C() { i = 3; }
}

privileged aspect A {
    after(C c) : executions(c.new(..)) {
        c.i = 4;
    }
}
```
Composition of Aspects

• Watch out!

• Aspect priority and domination

• Recursion -- aspects affecting themselves
Additional Notes

- Compiler Limitations
  - Throwing checked exceptions within advice of each \texttt{class()} 
  - preprocessing -- source level only! (for now) 
  - introducing \texttt{Serializable}