An introduction to Aspect-Oriented Programming with AspectJ

COMP 303
McGill University

Slides based on those from: Constantinos Constantinides

The AspectJ programming language

- AspectJ extends the Java programming language with constructs in order to support AOP.
- It is a superset of Java.
  - Each valid Java program is also a valid AspectJ program.
- It is a general-purpose language (as opposed to domain-specific).
- Currently the most notable AOP technology.

A first example: A bounded buffer

```
public class Buffer {
   private String[] BUFFER;
   int putPtr;   // keeps track of puts
   int getPtr;   // keeps track of gets
   int counter;  // holds number of items
   int capacity;
   Buffer (int capacity) {…}
   public boolean isEmpty() {…}
   public boolean isFull() {…}
   public void put (String s) {…}
   public String get() {…}
   }
}
```

- Class Buffer contains mutator and accessor methods:
  - Mutators: put(), get()
  - Accessors: isFull(), isEmpty()

Behavior of Buffer class

```
public class Buffer {
      …
      public void put (String s) {
        if (isFull())
          System.out.println("ERROR: Buffer full");
        else {
          BUFFER[putPtr++] = s;
          counter++;
        }
      }

      public String get() {
        if (isEmpty())
          return "ERROR: Buffer empty";
        else {
          counter--;
          return BUFFER[getPtr++];
        }
      }
}
```


AspectJ language concepts

- Joinpoint: a well-defined *event* in the execution of a program (such as the core functionality provided by class Buffer).
  - e.g. the call to method get() inside class Buffer.
- Pointcut: A collection of joinpoints.
  - e.g. the execution of all mutator methods inside class Buffer.
- Advice: A block of code that specifies some behavior to be executed before/after/around a certain joinpoint.
  - e.g. before the call to the body of method get(), display some message.

Example: Tracing

- Let us display a message before all calls to put() and get() inside Buffer.
- This pointcut specifies any call to put() in Buffer, taking a String argument, returning void, and with public access.

  call(public void Buffer.put(String))

- A *call* joinpoint captures an execution event after it evaluates a method calls' arguments, but before it calls the method itself.

Identifying joinpoints (cont.)

- This pointcut specifies all call events to get() in class Buffer, taking no arguments, returning String, and with public access:

```
call (public String Buffer.get())
```


Defining a pointcut

- We define a pointcut named “mutators” that combines both basic pointcut expressions.

```
pointcut mutators(): call(public void Buffer.put(String)) ||
                     call (public String Buffer.get());
```


Defining a pointcut (cont.)

- We may use logical operators in the definition of pointcuts in order to combine pointcut expressions:

1. || (OR operator)
   Matches a joinpoint if either the left pointcut expression matches or the right pointcut expression.
2. && (AND operator)
   Matches a joinpoint only when both the left pointcut expression and the right pointcut expression match.
3. ! (NOT operator)
   Matches all joinpoints not specfied by the pointcut

Define an advice

- An advice must be defined with respect to a pointcut, in this example we define an advice to mutators.
- This is a special type of advice, called “before advice”. As the term suggests, it specifies what must be done just before the event (joinpoint) specified by the pointcut.
- Pointcuts and advice together define composition (weaving) rules.

```
before(): mutators() {
        System.out.println("------ Mutator method called.");
}
```


Advice

- An advice associates the code to be executed with pointcuts.
- There are three ways to associate an advice with a pointcut:
  - Before: run just before the pointcut.
  - After: runs just after the pointcut.
    - May be after normal return, after throwing an exception or after returning either way from a joinpoint.
  - Around: Runs instead of the pointcut, with the provision for the pointcut to resume normal execution through proceed() (see later)

Providing an aspect definition

- Much like a class, an aspect is a unit of modularity.
- It is defined in terms of pointcuts (collections of joinpoints), advice, and ordinary Java fields and methods.
- Pointcuts say which events (joinpoints) to match, and the advice body says what to execute when it matches.

```
public aspect Tracer {
  pointcut mutators(): call(public void Buffer.put(String)) ||
                       call (public String Buffer.get());
  before(): mutators() {
    System.out.println("------ Mutator method called.");
  }
}
```

Tracing the execution (base program)

```
public class BufferDemo {
   public static void main(String[] args) {
      Buffer buffer = new Buffer(10);
      buffer.put("Hello");
      buffer.put("there");
      System.out.println(buffer.get());
      System.out.println(buffer.get());
   }
}
```

```
Hello
there
```


Tracing the execution (after weaving Tracer aspect)

```
public class BufferDemo {
   public static void main(String[] args) {
      Buffer buffer = new Buffer(10);
      buffer.put("Hello");
      buffer.put("there");
      System.out.println(buffer.get());
      System.out.println(buffer.get());
   }
}
```

```
------ Mutator method called.
------ Mutator method called.
------ Mutator method called.
Hello
------ Mutator method called.
there
```

```
public aspect Tracer {
   pointcut mutators(): call(public void Buffer.put(String)) ||
                        call (public String Buffer.get());
   before(): mutators() {
      System.out.println("------ Mutator method called.");
   }
}
```


Types of joinpoints

1. Calls to methods and constructors
2. Execution of methods and constructors
3. Field access
4. Exception handling
5. Class initialization
6. Lexical structure
7. Control flow
8. Self-target and argument-type
9. Conditional test

Patterns in pointcuts

- Pointcuts use a pattern language to specify events.
- The use of the * character is highly overloaded.
  - Using * where a type is expected matches any type.
  - Using * where an identifier is expected matches any identifier.
  - You can also use * within an identifier pattern.

Identifier Patterns

- the expression `foo*` would match any identifier starting with “foo”.
- the expression `*if*` would match any identifier with “if” in it.
- In general, an identifier pattern can by any valid identifier with * characters added to it.

Classname patterns

- Sometimes we wish to write patterns that identify a class or a set of classes.
- To identify a class in the default package we can just use an identifier pattern.
- We can string together identifier patterns to specify packages using “.” and “..”.
- We can add a “+” to the end of an identifier pattern to indicate that we wish to match a class and all of it subclasses.

Specifying classes

`foo` : class foo

`foo+` : class foo and all of its subclasses

`foo*` : all classes starting with “foo”

`*foo*` : all classes with “foo” in it

`foo*+` : all classes starting with “foo”, and all of their subclasses

Specifying packages and classes

- `MyPackage.foo` : the class foo in package MyPackage
- `MyPackage.*.foo` : the class foo that is in some immediate subpackage of MyPackage
- `MyPackage..foo` : the class foo that is in MyPackage or any subpackage of MyPackage.
- In package specifications “..”

  means . | .*. | .*.*. | …

Specifying arguments

- `(*)` : one argument, any type
- `(int)` : one argument of type int
- `(int,*)` : two arguments, first one with type int
- `()` : no arguments
- `(..)` : any number of arguments
- `(int,..)` : first argument of type int any number of other arguments

Specifying a method/constructor signature

Method:

[modifier_pattern] return_type_pattern
classtype_pattern.id_pattern (args_pattern)
[throws_pattern]

Constructor:

[modifier_pattern] return_type_pattern
classtype_pattern.new(args_pattern)
[throws_pattern]

Calls to methods and constructors

call (public static void MyClass.myMethod(String))	Call to public static myMethod() in MyClass taking a String argument, return type is void.
call (void MyClass.myMethod(..))	Call to myMethod() in MyClass taking any arguments, with void return type, and any access modifiers.
call (* MyClass.myMethod(..))	Call to myMethod() in MyClass taking any arguments, returning any type.
call (* MyClass.myMethod*(..))	Call to any method with name starting with “myMethod” in MyClass.

Calls to methods and constructors (cont.)

call (* MyClass.myMethod* (String,..))	Call to any method with name starting with “myMethod” in MyClass and the first argument is of String type.
call (* *.myMethod(..))	Call to myMethod() in any class in default package.
call (MyClass.new())	Call to the constructor of MyClass taking no arguments.
call (MyClass.new(..))	Call to the constructor of MyClass taking any arguments.

Calls to methods and constructors (cont.)

call (MyClass+.new(..))	Call to the constructor of MyClass or to the constructor of any of its subclasses, taking any arguments.
call (public * com.mycompany..*(..))	Call to all public methods in all classes in any package with com.company the root package.

Field access

- Capture read and write access to the fields of a class.
- The general format is
  get (FieldSignature) or set (FieldSignature)
- FieldSignature is
  [modifier_pattern] type_pattern field_pattern

get(PrintStream System.out)	Execution of read-access to field out of type PrintStream in System class.
set (int MyClass.x)	Execution of write-access to field x of type int in MyClass.

Exception handling

- Capture the execution of exception handlers of specified types.
- The general form is
  handler(ExceptionTypePattern)

handler (RemoteException)	Execution of catch-block handling RemoteException type
handler (IOException+)	Execution of catch-block handling IOException or its subclasses
handler (CreditCard*)	Execution of catch-block handling exception types with names that start with “CreditCard”.

Class initialization

- Capture the execution of static-class initialization (code specified in static blocks inside class definitions) of specified types.
- The general format is
  staticinitialization(TypePattern)

staticinitialization(MyClass)	Execution of static block of MyClass
staticinitialization(MyClass+)	Execution of static block of MyClass or its subclasses.

Lexical structure

- Capture joinpoints inside the lexical structure of class or a method.
- The general forms are
  within (TypePattern), or
  withincode(MethodOrConstructorSignature)

within(MyClass)	Any joinpoint inside the lexical scope of MyClass.
within(MyClass*)	Any joinpoint inside the lexical scope of classes with a name that starts with “MyClass”.
withincode(* MyClass.myMethod(..))	Any joinpoint inside the lexical scope of any myMethod() of MyClass.

Control flow

- Capture joinpoints based on the control flow of other joinpoints.
  - e.g. if a() calls b(), then b() is within the control flow of a().
- Take the forms
  cflow (joinpoint)
  cflowbelow(joinpoint)

cflow(call (* MyClass.myMethod(..))	All joinpoints in the control flow of a call to any myMethod() in MyClass including a call to the specified method itself.
cflowbelow(call (* MyClass.myMethod(..))	All joinpoints in the control flow of a call to any myMethod() in MyClass excluding a call to the specified method itself.

Self-target and argument-type

- Capture joinpoints based on self-obj, target-obj and arguments-type.

this(JComponent)	All joinpoints where this is instanceof JComponent
target(MyClass)	All joinpoints where the obj on which the method is called is of type MyClass.
args(String, …, int)	All joinpoints where the first argument is of type String and the last argument is of type int.
args(RemoteException)	All joinpoints where the type of argument or exception handler type is RemoteException

this/target/args

	this	target	args
method call	*caller*	*target*	*args*
const call	*caller*	-	*args*
method exec.	*this*	*this*	*args*
constr. exec.	*this*	*this*	*args*
handler	*this*	-	*exception*
get	*this*	*target*	*empty*
set	*this*	*target*	*value*

Conditional test

- Captures joinpoints based on some conditional check at the joinpoint.
- Takes the form
  if (BooleanExpression)

if (EventQueue.isDispatchedThread())	All joinpoints where EventQueue.isDispatchedThread() evaluates to true.

Reflection and thisJoinPoint

- With reflection we can examine information at an execution point (joinpoint).
- Each advice has access to **thisJoinPoint** which contains information about the joinpoint.
- Can also use **thisJoinPointStaticPart** to get only static information (this may be less expensive)

Example: Tracing with reflection

- Let us trace the execution of all methods inside class Buffer, with any type of arguments, returning any type and with any access type.
- The pointcut below specifies the above events:
  execution (* Buffer.*(..))
- This is an example of a named pointcut:
  pointcut publics(): execution (* Buffer.*(..));
- An advice may also use an unnamed pointcut:
  before(): execution (* Buffer.*(..)) {…}

A Tracing aspect with reflection

```
public aspect ReflectionTracer {
   pointcut publics(): execution (* Buffer.*(..));
   before(): publics() {
      System.out.println("Before: " + thisJoinPoint);
   }
   after(): publics() {
      System.out.println("After: " + thisJoinPoint);
   }
}
```

Running the tracing example

```
public class BufferDemo {
  public static void main(String[] args) {
    Buffer buffer = new Buffer(10);
    buffer.put("Hello");
    buffer.put("there");
    System.out.println(buffer.get());
    System.out.println(buffer.get());
  }
}
```

```
public aspect ReflectionTracer {
  pointcut publics(): execution (* Buffer.*(..));
  before(): publics() {
    System.out.println("Before: " + thisJoinPoint);
  }
  after(): publics() {
    System.out.println("After: " + thisJoinPoint);
  }
}
```

```
Before: execution(void Buffer.put(String))
Before: execution(boolean Buffer.isFull())
After: execution(boolean Buffer.isFull())
After: execution(void Buffer.put(String))
Before: execution(void Buffer.put(String))
Before: execution(boolean Buffer.isFull())
After: execution(boolean Buffer.isFull())
After: execution(void Buffer.put(String))
Before: execution(String Buffer.get())
Before: execution(boolean Buffer.isEmpty())
After: execution(boolean Buffer.isEmpty())
After: execution(String Buffer.get())
Hello
Before: execution(String Buffer.get())
Before: execution(boolean Buffer.isEmpty())
After: execution(boolean Buffer.isEmpty())
After: execution(String Buffer.get())
there
```


Type modification constructs

- Using an advice we are able to affect the dynamic behavior of a system.
- Sometimes it is necessary to provide aspectual behavior over the static structure of the system.
- AspectJ allows a number of static-crosscutting types, including:
  - Introduction of new methods and fields.
  - Introduction of supertypes.

Example: Providing timestamp behavior to Buffer class

- Introducing a private variable named timestamp of type long to Buffer class:

```
private long Buffer.timestamp;
```

- Introducing a void method in Buffer to set the timestamp:

```
public void Buffer.timestamp() {
  // "this" refers to Buffer class and not to Timestamp aspect
  this.timestamp = System.currentTimeMillis();
}
```


Example: Providing timestamp behavior to Buffer class (cont.)

- Introducing a long method in Buffer to return the timestamp:

```
public long Buffer.getTimestamp() {
  return timestamp;
}
```


Example: Providing timestamp behavior to Buffer class (cont.)

```
public aspect Timestamp {
  private long Buffer.timestamp;

  public long Buffer.getTimestamp() {
    return timestamp;
  }

  public void Buffer.timestamp() {
    // "this" refers to Buffer class and not to Timestamp aspect
    this.timestamp = System.currentTimeMillis();
  }

}
```


Introduction of supertypes

- Introducing a supertype to one or more class, affects the inheritance hierarchy of the system.
- We can declare superclasses and interfaces to an existing class or interface.

Example

- Let us introduce a TimestampedObject interface to Buffer class.
- Consider the following interface that defines getTimestamp() and timestamp():

```
public interface TimestampedObject {
        long getTimestamp();
        void timestamp();
}
```


Introducing field and methods

- The field and method introduction in the aspect definition would now refer to the interface, not the Buffer class:

```
private long TimestampedObject.timestamp;

public long TimestampedObject.getTimestamp() {
        return timestamp;
}

public void TimestampedObject.timestamp() {
        // "this" refers to Buffer class and not to Timestamp aspect
        this.timestamp = System.currentTimeMillis();
}
```


Declare an interface to class Buffer

- The aspect can now dictate that class Buffer should implement the Timestamp interface:

```
declare parents: Buffer implements TimestampedObject;
```


Exposing context: pointcut

- Pointcuts can expose part of the execution context at the joinpoints.
- Values exposed by a pointcut can be used in the body of an advcice declaration.
- The pointcut exposes and publishes one value, namely a reference to the Buffer instance

```
pointcut bufferChanged(Buffer obj):
        execution (* Buffer.*(..)) &&
        this(obj);
```


Exposing context: advice

- An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses.

```
after(TimestampedObject obj): bufferChanged(obj) {
  obj.timestamp();
  System.out.println("Operation " + thisJoinPoint + " at " +
          obj.getTimestamp());
}
```


Putting everything together

```
public aspect Timestamp {
  private long TimestampedObject.timestamp;
  public long TimestampedObject.getTimestamp() {
    return timestamp;
  }
  public void TimestampedObject.timestamp() {
    // "this" refers to Buffer class and not to Timestamp aspect
    this.timestamp = System.currentTimeMillis();
  }
  declare parents: Buffer implements TimestampedObject;
  pointcut bufferChanged(Buffer obj): execution (* Buffer.*(..)) && this(obj);
  after (TimestampedObject obj): bufferChanged(obj) {
    obj.timestamp();
    System.out.println("Operation " + thisJoinPoint + " at " + obj.getTimestamp());
  }
}
```

Running the application

```
Operation execution(boolean Buffer.isFull()) at 1096152607327
Operation execution(void Buffer.put(String)) at 1096152607327
Operation execution(boolean Buffer.isFull()) at 1096152607327
Operation execution(void Buffer.put(String)) at 1096152607327
Operation execution(boolean Buffer.isEmpty()) at 1096152607327
Operation execution(String Buffer.get()) at 1096152607327
Hello
Operation execution(boolean Buffer.isEmpty()) at 1096152607337
Operation execution(String Buffer.get()) at 1096152607337
there
```


Around advice

- The third type of advice, around(), gives a chance to affect whether and when the joinpoint (event) is executed, using the special proceed() syntax.

Example: Providing contract checking to the Buffer class

- Provide a new Buffer class:

```
public class BBuffer {
    private String[] BUFFER;
    private int putPtr;                 // keeps track of puts
    private int getPtr;                 // keeps track of gets
    protected int capacity;

    BBuffer (int capacity) {
        BUFFER = new String[capacity];
        this.capacity = capacity;
    }

    public void put (String s) {BUFFER[putPtr++] = s;}

    public String get() {return BUFFER[getPtr++];}

}
```


Introducing state to BBuffer and declaring pointcuts

```
private int BBuffer.counter = 0;

private boolean BBuffer.isEmpty() {
    return (this.counter==0);
}

private boolean BBuffer.isFull() {
    return (this.counter == this.capacity);
}

pointcut puts(BBuffer obj):
                execution (* BBuffer.put(String)) && this(obj);

pointcut gets(BBuffer obj):
                execution (* BBuffer.get()) && this(obj);
```


around() advice for puts()

```
void around (BBuffer obj): puts(obj) {
    if (obj.isFull())
        System.out.println("ERROR: Buffer full");
    else {
        // go ahead with the method call.
        // proceed() takes the same number and types of arguments
        // as the around() advice.
        proceed(obj);
        obj.counter++;
    }
}
```


around() advice for gets()

```
String around(BBuffer obj) : gets(obj){
    if (obj.isEmpty())
        return "ERROR: Buffer empty";
    else {
        obj.counter--;
        return proceed(obj);
    }
}
```

Synchronization aspect

```
public aspect Synchronization {
    private int BBuffer.counter = 0;
    private boolean BBuffer.isEmpty() {return (this.counter==0);}
    private boolean BBuffer.isFull() {return (this.counter == this.capacity);}
    pointcut puts(BBuffer obj): execution (* BBuffer.put(String)) && this(obj);
    pointcut gets(BBuffer obj): execution (* BBuffer.get()) && this(obj);
    void around (BBuffer obj): puts(obj) {
        if (obj.isFull()) System.out.println("ERROR: Buffer full");
        else {
            // go ahead with the method call.
            // proceed() takes the same number and types of arguments
            // as the around() advice.
            proceed(obj);
            obj.counter++;
        }
    }
    String around(BBuffer obj) : gets(obj){
        if (obj.isEmpty()) return "ERROR: Buffer empty";
        else {
            obj.counter--;
            return proceed(obj); }}}
```


Running the application

```
public class BufferDemo {
   public static void main(String[] args) {
      BBuffer buffer = new BBuffer(2);
      buffer.put("Item 1 ");
      buffer.put("Item 2 ");
      buffer.put("Item 3 ");
      buffer.put("Item 4 ");
      System.out.println(buffer.get());
      System.out.println(buffer.get());
      System.out.println(buffer.get());
      System.out.println(buffer.get());
   }
}
```

```
ERROR: Buffer full
ERROR: Buffer full
Item 1
Item 2
ERROR: Buffer empty
ERROR: Buffer empty
```


Privileged aspects

- You can mark an aspect as ‘privileged’ which would give it access to the private features of the affected class(es).
- In the previous example, capacity had protected access to enable isFull() to get access to it.

```
public class BBuffer {
   …
   protected int capacity;
…
}
```

```
private boolean BBuffer.isFull() {
    return (this.counter == this.capacity);
}
```


Privileged aspects

- We can redefine capacity as private and mark the synchronization aspect as privileged.

```
public class BBuffer {
   …
   private int capacity;
…
}
```

```
privileged public aspect Synchronization {…}
```


Determining precedence among advice

- Multiple pieces of advice may apply to the same pointcut.
- In this case, the resolution order of the advice is based on rules on advice precedence.
- There are two cases:
  1. Precedence rules among advice from different aspects.
  2. Precedence rules among advice from within the same aspect.

Precedence rules among advice from different aspects

1. If aspect A is declared to have precedence over aspect B, then all advice in (concrete) aspect A has precedence over all advice in (concrete) aspect B when they are on the same join point.
2. Otherwise, if aspect A is a subaspect of aspect B, then all advice defined in A has precedence over all advice defined in B. So, unless otherwise specified with declare precedence, advice in a subaspect has precedence over advice in a superaspect.
3. Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.

Example

```
public class C {
    public static void main(String[] args) {
        System.out.println("Inside main");
    }
}
```

Before from A
Before from B
Inside main
After from B
After from A

```
public aspect A {
    declare precedence: A, B;
    pointcut callMain(): execution (public static void C.main(..));
    before(): callMain() {System.out.println("Before from A");}
    after(): callMain() {System.out.println("After from A");}
}
```

```
public aspect B {
    pointcut callMain(): execution (public static void C.main(..));
    before(): callMain() {System.out.println("Before from B");}
    after(): callMain() {System.out.println("After from B");}
}
```


Precedence rules among advice from the same aspect

- If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, the one that appears earlier in the aspect has precedence over the one that appears later.

Example 1

```
public aspect E {
    pointcut callMain(): execution (public static void C.main(..));

    after(): callMain() {
        System.out.println("After from E");
    }

    after(): callMain() {
        System.out.println("After from E - placed below");
    }

}
```

Inside main
After from E
After from E - placed below

Example 2

```
public aspect D {
    pointcut callMain(): execution (public static void C.main(..));

    before(): callMain() {
        System.out.println("Before from D - placed above");
    }

    before(): callMain() {
        System.out.println("Before from D - placed below");
    }

}
```

Before from D - placed above
Before from D - placed below
Inside main

Abstract aspects example

```
public abstract aspect AbstractTracer {
    abstract pointcut logPoints();

    before(): logPoints() {
        System.out.println("Entering: " + thisJoinPoint);
    }

    after(): logPoints() {
        System.out.println("Exiting: " + thisJoinPoint);
    }
}
```

```
public aspect TraceMethods extends AbstractTracer {
    pointcut logPoints(): call (* Buffer.*(..));
}
```


References

- AspectJ on-line user's guide, available from www.eclipse.org/aspectj
- Ramnivas Laddad, "I want my AOP" (Parts I, II), JavaWorld.
- Nicholas Lesiecki, "Improve modularity with aspect-oriented programming", IBM DeveloperWorks.