The building blocks of AspectJ: Join points, pointcuts and advices

- We will follow a bottom-up approach by discussing the building blocks of AspectJ.
- Consider the implementation of an unbounded stack. The stack implements a last-in-first-out (LIFO) protocol.
- Variable stack is an ArrayList that contains a collection of elements.
- Variable top holds the current size of the stack, initialized to −1 implying that the collection is empty.

private ArrayList<String> stack = new ArrayList<String> ();
protected int top = -1;
The building blocks of AspectJ /cont.

- The interface of class Stack contains a number of methods.
- We can distinguish between those methods that modify the state of an object, referred to as *mutators*, and those that access the state but do not modify it, referred to as *accessors*.
- Methods `push()` and `pop()` are mutators, whereas methods `top()`, `isEmpty()` and `size()` are accessors.

```java
public void push (String str) {...}
public String pop () {...}
public String top () {...}
protected boolean isEmpty () {...}
public int size () {...}
```

Join points

- A *join point* is a point in the execution of the program.
- We can regard join points as events of interest that can be captured by the underlying language.
- AspectJ supports a rich set of join points that includes *message sends* and *execution of methods*.
- In this example, we want to capture all `push` and `pop` messages sent to an object of type `Stack`. 
The following join point

call(void Stack.push(String))

captures a push message that includes one argument of type String, sent to an object of type Stack, where the invoked method is not expected to return any value.

Note that in the literature the expression is interpreted in terms of a call to a method as follows: The join point captures a call to method push() in class Stack, taking a String argument and returning no value (void). The modifier of the method is not specified, implying that it can be of any type.

Similarly the following join point

call(String Stack.pop())

captures a pop message that includes no argument, sent to an object of type Stack, where the receiver object is expected to return a value of type String.
Pointcuts

- Since we want to log both push and pop messages, we can combine the two join points into a single disjunctive expression.
- A pointcut (or pointcut designator) is a logical expression composed by individual join points.
- Additionally, a pointcut may be given and it can subsequently be referred to by an identifier.

Pointcuts /cont.

- Consider pointcut mutators() that combines the two individual join points into a logical disjunction as follows:
  
  ```java
  pointcut mutators(): call(void Stack.push(String)) || call(String Stack.pop());
  ```

- Since a join point refers to an event, we say that a join point is captured whenever the associated event occurs.
- Consequently, we say that a pointcut is captured whenever the logical expression made up of individual join points becomes true.
Advices

▶ In this example once a push or pop message is sent, and before any corresponding method executes, we want to first display some message.

▶ An advice is a method-like block, that associates to a pointcut, defining behavior to be executed.

▶ However, unlike a method, an advice block is never explicitly called. Instead, it is only implicitly invoked once its associated pointcut is captured.

Advices /cont.

▶ The following advice

```java
before(): mutators() {
    System.out.println(">Message sent to update stack.");
}
```

is attached to pointcut mutators().

▶ Once a push() or pop() message is sent to an object of type Stack, the pointcut mutators() is captured. Before the message can proceed, the before advice will execute.
Advices /cont.

- AspectJ provides a level of granularity which specifies exactly when an advice block should be executed, such as executing *before*, *after*, or *instead of* the code that is associated with the pointcut.

- More specifically, an advice block can be:
  - *before*: An advice that runs before the code associated with the pointcut expression.
  - *after*: An advice that runs after the code associated with the pointcut expression (It may be after normal return, after throwing an exception or after returning either way from a join point).

Advices /cont.

- *around*: An advice that runs instead of the code associated with the pointcut expression, with the provision for the pointcut to resume normal execution through a `proceed` call (see later).
Named and unnamed pointcuts

- In the example above, mutators() is a named pointcut. As the term suggests, it is an expression bound to an identifier.

```java
pointcut mutators(): call(void Stack.push(String)) ||
    call(String Stack.pop());
before(): mutators() {
    System.out.println(">Message sent to update stack.");
}
```

- On the other hand, an unnamed (or anonymous) pointcut is an expression not bound to an identifier but instead it is directly attached to an advice as shown below:

```java
before(): call(void Stack.push(String)) ||
    call(String Stack.pop()); {
    System.out.println(">Message sent to update stack.");
}
```

- The two pointcuts are semantically equivalent.

Putting everything together: An aspect definition

- Much like a class, an aspect is a unit of modularity.
- We can now provide an aspect definition as follows:

```java
public aspect Logger {
    pointcut mutators(): call(void Stack.push(String)) ||
        call(String Stack.pop());
    before(): mutators() {
        System.out.println(">Message sent to update stack.");
    }
}
```
Running the program

```java
Stack myStack = new Stack();
myStack.push("base");
myStack.push("your");
myStack.push("all");
System.out.println(myStack.pop()); // Message sent to update stack.
System.out.println(myStack.pop()); // Message sent to update stack.
System.out.println(myStack.pop()); // Message sent to update stack.
System.out.println(myStack.top()); // Message sent to update stack.
```

Dissection of a pointcut

- In the previous example, we had defined a named pointcut as follows:

  ```java
  pointcut mutators(): call(void Stack.push(String)) ||
  call(String Stack.pop());
  ```

- The format of a named pointcut is

  ```java
  pointcut <name> ([$<object(s) to be picked up>]): <join point expression>
  ```

  where a join point expression is any predicate over join points.
Dissection of a join point

- A join point has the following format:
  <join point type> (<signature>)

- In the above example, pointcut mutators() was defined as the logical disjunction of two join points, both of type call, and picked up no object (see later on context passing).
Call join points

- A *call join point* captures a message that matches a given signature that is sent to an object with a given static type.
- For example, the join point `call (void Server.attach(..))` captures message `attach` with any (including zero) arguments sent to an object whose static type is `Server` and where the invoked method is not expected to return any value.
The format of a call join point is

\[ \text{call} \ (\text{signature}) \]

where the format of \textit{signature} is

\[ [<\text{modifier}>] \ <\text{return type}> \ <\text{class}.<\text{method}>(<\text{parameter list}>) \]
Join point signatures: Examples

- The signature below

  protected void Vector.removeRange(int, int)

captures a protected void method removeRange in class Vector, taking two arguments of type int.
- We can use + to include subclasses of Vector.

  protected void Vector+.removeRange(int, int)

Join point signatures: Examples /cont.

- The signature below

  * * Vector.removeElement(Object)

captures method removeElement of any return type and any modifier in class Vector, taking one argument of type Object.
- As the modifier is optional it can be omitted.

  * Vector.removeElement(Object)

- The signature below

  * * *.removeElement(Object)

captures method removeElement of any return type and any modifier in any class, taking one argument of type Object.
Join point signatures: Examples /cont.

- The signature below
  
  * * *.*(int, ..)

  captures any method in any class, returning any type, of any modifier, taking an argument of type int, followed by a sequence of additional arguments (that can also be empty) of any type.

- Similarly, the signature below
  
  * * *.*(.., int)

  captures any method in any class, returning any type, of any modifier, taking an argument of type int preceded by a sequence of additional arguments (that can also be empty) of any type.

Join point signatures: Examples /cont.

- The signature below
  
  * * *.*(..)

  captures any method in any class, returning any type, of any modifier, taking any (or zero) arguments.

- As above but taking no arguments:
  
  * * *.*()

- The signature below
  
  * * Vector.remove*(..)

  captures any method whose name starts with the string remove followed by zero or more characters, of any return type and any modifier, in class Vector, taking any (or zero) arguments.
Join point signatures: Examples /cont.

- If we used this join point (as an anonymous pointcut) in the Logger aspect in the Stack example, i.e.
  
  ```java
  before(): call (* Stack.*(..)) {
    System.out.println(">Message sent to update stack.");
  }
  ```

  then the pointcut would also capture messages `isFull()` and `isEmpty()` as well as any other messages sent to a Stack object whether existing or introduced in the future, such as `toString()`, `clone()`, etc.

---

Call to constructor join points

- AspectJ distinguishes between regular messages (sent to objects and classes) and messages that are sent to class constructors.
- A *call to constructor join point* captures a call made to the constructor of a given class.
- The keyword `new` is used to identify such join point signatures.
- For example, `call(Stack.new())` captures a call to the default constructor of class `Stack`.
- Note that signatures of constructor call join points contain no return type.
Call to constructor join points /cont.

- The format of a call to constructor join point is
  \[
  \text{call} \ (\text{signature})
  \]
  where the format of \textit{signature} is
  \[
  [\langle \text{modifier} \rangle] \ \langle \text{class} \rangle.\text{new}(\langle \text{parameter list} \rangle)
  \]

Call join points in the presence of inheritance

- A call join point captures a message that is sent to an object of a given static type. This implies that it can capture such message sent to an object of a subtype.

- We subclassify \textit{Stack} to define class \textit{BStack} that implements a stack of bounded capacity.

  public class BStack extends Stack {
    private int capacity;
    public BStack (int capacity) {
      this.capacity = capacity;
    }
    @Override
    public void push (String str) {
      if (!this.isFull())
        super.push(str);
    }
    private boolean isFull() {
      return top == capacity;
    }
Running the program

BStack myStack = new BStack(2);
myStack.push("base");
myStack.push("your");
myStack.push("all");
System.out.println(myStack.pop());
System.out.println(myStack.pop());
System.out.println(myStack.pop());
System.out.println(myStack.top());

Running the program: Observations

▶ We see that pointcut mutators() is captured.
▶ The reason is that the call join point

    call(void Stack.push(String))

captures calls to push(String) declared in class Stack or any of its subclasses.
Reflective information on join points with thisJoinPoint

- AspectJ provides the special variable thisJoinPoint that contains reflective information about the current join point.
- Let us modify aspect Logger, in the bounded stack example, to access reflective information on all join points captured by mutators():

```java
public aspect Logger {
    pointcut mutators(): call(void Stack.push(String)) ||
                        call(String Stack.pop());
    before(): mutators() {
        System.out.println(">Message sent to update stack: " +
                          thisJoinPoint);
    }
}
```

Reflective information on join points with thisJoinPoint /cont.

- We run the test program on the bounded stack and we see that it has the same behavior as before, but with additional information on each captured join point:

```
>Message sent to update stack: call(void BStack.push(String))
>Message sent to update stack: call(void BStack.push(String))
>Message sent to update stack: call(void BStack.push(String))
>Message sent to update stack: call(String BStack.pop())
 your null
>Message sent to update stack: call(String BStack.pop())
 base null
>Message sent to update stack: call(String BStack.pop())
null
```
Multiple pointcuts

- A pointcut may capture a set of join points. This implies that two (or more) pointcuts may share join points.
- Consider the following example:

```
public class Human {
    public String reason() {
        return "I am a human and I can reason.";
    }
}

public class Bladerunner extends Human { }
```

Multiple pointcuts /cont.

- Aspect Logger defines the following:

```
    call(String Human.reason()) captures messages reason() sent to objects of type Human:

    before() : call(String Human.reason()) {
        System.out.println("> Captured call to Human.reason(): " +
                          thisJoinPoint);
    }
```
Multiple pointcuts /cont.

- call(String Bladerunner.reason()) captures messages reason() sent to objects of type Bladerunner:

  before() : call(String Bladerunner.reason()) {
    System.out.println(">Captured call to Bladerunner.reason(): " +
    thisJoinPoint);
  }

Multiple pointcuts /cont.

- Consider the code below:

  Human sebastian = new Human();
  System.out.println(sebastian.reason());

- The statement sebastian.reason() is captured by pointcut call(String Human.reason()).

- The body of the associated advice will execute, displaying

  >Captured call to Human.reason(): call(String Human.reason())
Consider the code below:

```java
Bladerunner deckard = new Bladerunner();
System.out.println(deckard.reason());
```

The statement `deckard.reason()` will be captured by both pointcuts.

Both advices will execute, displaying

```
> Captured call to Human.reason(): call(String Bladerunner.reason())
> Captured call to Bladerunner.reason():
    call(String Bladerunner.reason())
```

### Execution join points

- As its name suggests, an *execution join point* captures the execution of a method defined in a given class.
- The patterns shown for call join points are also applicable to execution join points.
- For example, the join point

  ```java
  execution (void Server.attach(..))
  ```

  captures the execution of void method `attach` with any (including zero) parameters defined in class `Server`, regardless of its visibility.
Execution join points /cont.

▶ The format of an execution join point is

\[ \text{execution} (\text{signature}) \]

where the format of \text{signature} is

\[ [<\text{modifier}>] \ <\text{return type}> \ <\text{class}>.<\text{method}>(<\text{parameter list}>) \]

Execution join points /cont.

▶ We have slightly modified aspect \text{Logger} from the previous example by replacing the call join points by execution join points and adjusting the messages displayed:

public aspect Logger {
   before() : execution(String Human.reason()) {
      System.out.println(">Captured execution of " +
      "Human.reason(): " +
      thisJoinPoint);
   }
   before() : execution(String Bladerunner.reason()) {
      System.out.println(">Captured execution of " +
      "Bladerunner.reason(): " +
      thisJoinPoint);
   }
}

▶ It is important to see that the pointcut \text{execution(String Bladerunner.reason())} will never be captured as there exists no method \text{reason()} defined in class \text{Bladerunner}. 
Execution join points /cont.

▶ For the same test program

```java
Human sebastian = new Human();
Bladerunner deckard = new Bladerunner();
System.out.println(sebastian.reason());
System.out.println(deckard.reason());
```

the run-time system will invoke method reason() defined in class Human twice.

▶ The output is as follows:

```plaintext
> Captured execution of Human.reason(): execution(String Human.reason())
I am a human and I can reason.
> Captured execution of Human.reason(): execution(String Human.reason())
I am a human and I can reason.
```

Constructor execution join points

▶ AspectJ distinguishes between executions of regular methods and executions of constructor methods.

▶ Executions of constructor methods are identified in join point signatures by the keyword `new`.

▶ Note also that join point signatures of constructor executions contain no return type.

▶ The format of a constructor execution join point is

```
execution (signature)
```

where the format of `signature` is

```
[<modifier>] <class>.new(<parameter list>)
```
Call vs. execution join points: An example

▶ Consider classes Server and Client:

```java
public class Server {
    public String ping() {
        System.out.println("Inside Server.ping().");
        return "pong.";
    }
}

public class Client {
    Server server;
    public Client(Server server) {
        this.server = server;
    }
    public String testConnection() {
        System.out.println("About to call server.ping()");
        String str = server.ping();
        System.out.println("Just called server.ping()");
        return str;
    }
}
```

Call vs. execution join points: An example /cont.

▶ Aspect Logger captures calls to String Server.ping() and executions of String Server.ping():

```java
public aspect Logger {
    before() : call(String Server.ping()) {
        System.out.println(">Before: " + thisJoinPoint);
    }
    after() : call(String Server.ping()) {
        System.out.println(">After: " + thisJoinPoint);
    }
    before() : execution(String Server.ping()) {
        System.out.println(">Before: " + thisJoinPoint);
    }
    after() : execution(String Server.ping()) {
        System.out.println(">After: " + thisJoinPoint);
    }
}
```
Consider the test program below and its output:

```java
Server server = new Server();
Client client = new Client(server);
System.out.println(client.testConnection());
```

About to call `server.ping()`
>Before: call(String Server.ping())
>Before: execution(String Server.ping())
Inside `Server.ping()`.
>After: execution(String Server.ping())
>After: call(String Server.ping())
Just called `server.ping()`
pong.
Lexical structure join points

- Lexical structure join points capture well-defined points inside the lexical structure of classes or methods.
- The forms are

\[
\text{within (type pattern)} \\
\text{withincode (method signature)}
\]

where type pattern may include wildcard characters and must resolve to a class, or a range of different classes, and method signature may include wildcard characters and must resolve to a method in a class or to a range of methods.

Control flow join points

- The term control flow refers to the order in which events, such as messages or method executions, occur.
- For example, if during event\(_1\) the event causes event\(_2\) which in turn causes event\(_3\), then we say that the sequence \(\langle \text{event}_2, \text{event}_3 \rangle\) lies within the control flow of event\(_1\), as well as \(\langle \text{event}_3 \rangle\) lies within the control flow of event\(_2\).
Control flow join points /cont.

- Recall that in AspectJ, events are captured by join points and pointcuts.
- The format of a control flow join point is

\[ \text{cflow} (\text{pointcut designator}) \]

where \text{pointcut designator} can be any pointcut.

- Note that \text{cflow} (\text{pointcut}) captures \text{pointcut} itself as well as all subsequent pointcuts in its control flow.

- If we want to exclude \text{pointcut} and capture only those pointcuts that occur subsequently, we must use the following alternative notation:

\[ \text{cflowbelow} (\text{pointcut designator}) \]

Control flow join points: An example

- Let us now consider the hierarchy in the Bladerunner example, where aspect Logger contains two before advices on the executions of \text{Human.reason()} and \text{Bladerunner.reason()}. The aspect is shown here again:

```java
public aspect Logger {
    before() : execution(String Human.reason()) {
        System.out.println(">Captured execution of " +
                         "Human.reason(): " +
                         thisJoinPoint);
    }
    before() : execution(String Bladerunner.reason()) {
        System.out.println(">Captured execution of " +
                         "Bladerunner.reason(): " +
                         thisJoinPoint);
    }
}
```
We will proceed to add a new aspect to the project.

In ReflectiveLogger the unnamed pointcut of the after advice captures all method executions made in the program, but not those made from within itself and not those within the control flow of the Java system, i.e. execution of library methods.

```java
public aspect ReflectiveLogger {
    after(): execution(* *(..))
    && !within(ReflectiveLogger)
    && !cflow(execution (* java.*.*.*(..))) {
        System.out.println(">Executed: " + thisJoinPoint);
    }
}
```

```java
Human sebastian = new Human();
System.out.println(sebastian.reason());

>Captured execution of Human.reason():
    execution(String Human.reason())

>Executed: execution(String Human.reason())
I am a human and I can reason.
```
Control flow join points: An example /cont.

Bladerunner deckard = new Bladerunner();
System.out.println(deckard.reason());

>Captured execution of Human.reason():
    execution(String Human.reason())
>Executed: execution(String Human.reason())
I am a human and I can reason.
>Executed: execution(void Test.main(String[]))

Field access join points

- Field access join points capture read and write access to the fields declared in a given class.
- The format is

  get (field signature)
  set (field signature)

for read and write access respectively, where a field signature is defined as

  [[<modifier>]] <return type> <class>..<field>

- A field signature may contain wildcard characters and it must resolve to an attribute of a given class.
Field access join points: An example

```java
public class GPSCoordinate {
    private Double latitude = 0.;
    private Double longitude = 0.;
    public GPSCoordinate(Double latitude, Double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    public void setLatitude(Double latitude) {
        this.latitude = latitude;
    }
    public Double getLatitude() {
        return this.latitude;
    }
    public void setLongitude(Double longitude) {
        this.longitude = longitude;
    }
    public Double getLongitude() {
        return this.longitude;
    }
    public void moveTo(Double latitude, Double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    public String toString() {...}
}
```

Field access join points: An example /cont.

- Aspect `FieldAccess` defines two unnamed pointcuts to capture read and write access, respectively, to any field of class `GPSCoordinate`.
- Note that the join points are able to capture access to the fields despite the fact that the fields are declared `private`.
- Note also that these join points do not capture inherited fields.

```java
public aspect FieldAccess {
    before() : get(* GPSCoordinate.*) {
        System.out.println(">Read access: " + thisJoinPoint);
    }
    before() : set(* GPSCoordinate.*) {
        System.out.println(">Write access: " + thisJoinPoint);
    }
}
```
We create and initialize an instance of `GPSCoordinate`:

```java
GPSCoordinate point = new GPSCoordinate(45.220227, -73.564453);
```

```java
>Write access: set(Double GPSCoordinate.latitude)
>Write access: set(Double GPSCoordinate.longitude)
>Write access: set(Double GPSCoordinate.latitude)
>Write access: set(Double GPSCoordinate.longitude)
```

We move the object to a new location and finally it displays its latitude and longitude values. The output of the program is shown below:

```java
point.moveTo(46.763321, -71.224365);
```

```java
>Write access: set(Double GPSCoordinate.latitude)
>Write access: set(Double GPSCoordinate.longitude)
```
Field access join points: An example /cont.

▶ We display the object’s latitute and longitude values:

```java
System.out.println(point.getLatitude());
System.out.println(point.getLongitude());
```

> Read access: get(Double GPSCoordinate.latitude)
46.763321
> Read access: get(Double GPSCoordinate.longitude)
-71.224365

Field access join points: An example /cont.

▶ We display the object’s state via a call to toString():

```java
System.out.println(point.toString());
```

> Read access: get(Double GPSCoordinate.latitude)
> Read access: get(Double GPSCoordinate.longitude)
(46.763321, -71.224365)
Around advice

- The third type of advice allows us to say “Whenever a pointcut is captured, instead of running the code associated with the pointcut, execute the body of the advice.”
- An optional mechanism allows us to resume execution of the code associated with the pointcut.

Visualization of around advice
public class Buffer {
    String[] BUFFER;
    int putPtr;
    int getPtr;
    int counter;
    int capacity;
    String name;
    Buffer (int capacity) {
        BUFFER = new String[capacity];
        this.capacity = capacity;
    }
    Buffer (int capacity, String name) {
        this(capacity);
        this.name = name;
    }
    public String getName() {
        return name;
    }
    private boolean isEmpty() {
        return (counter == 0);
    }
    private boolean isFull() {
        return (counter == capacity);
    }
    public void put (String s) {
        if (isFull())
            System.out.println("ERROR: Buffer full");
        else {
            BUFFER[putPtr++ % (capacity)] = s;
            counter++;
        }
    }
    public String get() {
        if (isEmpty())
            return "ERROR: Buffer empty";
        else {
            counter--;
            return BUFFER[getPtr++ % (capacity)];
        }
    }
}

public class Buffer2 extends Buffer {
    Buffer2 (int capacity) {
        super(capacity);
    }
    public String gget() {
        return super.get();
    }
}

▶ Consider now the definition of class Buffer2 which introduces method gget().

▶ This method behaves exactly like get(), but it can only execute after a get().

▶ To implement this requirement in Java would imply that Buffer2 would have to re-define methods put() and get() to implement a history protocol. Instead, we will implement the history protocol in an aspect.
In defining the aspect, we first need to introduce a variable to serve as a flag that would indicate which operation has been lastly executed:

```java
private boolean afterGet;
```

We must update the history flag `afterGet` appropriately after the execution of both `put()` and `get()`. Note that both methods have been inherited to `Buffer2`. Thus, if we say

```java
execution(void Buffer2.put(String))
```

then this join point would never be caught, since the method is defined (and therefore executes) from within `Buffer`.

We can capture proper behavior as follows:

```java
after(): execution(void Buffer.put(String)){
    afterGet = false;
}
after(): execution(String Buffer.get()) {
    afterGet = true;
}
```
We now need to write code to say: “Once there is a message get() sent to an object of type Buffer2, instead of running the code that should run, check the history of method execution and if the previous executed method was a get(), then allow execution to go ahead; Otherwise, issue an error.”

We use the around advice for this as shown below.

```java
String around() : call (String Buffer2.gget()) {
    if (afterGet == false)
        return "ERROR: Cannot execute gget()";
    else {
        return proceed();
    }
}
```

The call to `proceed` allows execution to resume at the code associated with the pointcut.

One thing to remember is that unlike the two other types of advices, before and after, the around advice must contain a return type which should be the same as the return type of the method of the associated pointcut.

```java
public String gget() {...}
```

In this example, as the pointcut is on `gget()` with a return type `String`, then the same return type must be associated with the advice.

If there is more than one method invoked with different return types, then the type of the around advice should be `Object`.

```java
String around() : call (String Buffer2.gget()) {...}
```
We can put everything in one aspect definition as follows:

```java
public aspect HistoryProtocol {
    private boolean afterGet;
    after(): execution(void Buffer.put(String)) {
        afterGet = false;
    }
    after(): execution(String Buffer.get()) {
        afterGet = true;
    }
    String around() : call (String Buffer2.gget()) {
        if (afterGet == false)
            return "Error: Cannot execute gget()";
        else
            return proceed();
    }
}
```

Buffer2 buffer = new Buffer2(5);
buffer.put("all");
buffer.put("your");
buffer.put("base");
buffer.put("are");
buffer.put("belong");
System.out.println(buffer.gget());
System.out.println(buffer.get());
System.out.println(buffer.gget());
buffer.put("to");
buffer.put("us");
System.out.println(buffer.gget());
System.out.println(buffer.get());
System.out.println(buffer.gget());
System.out.println(buffer.get());
System.out.println(buffer.get());
Advice precedence

▶ Several advice blocks may apply to the same join point.
▶ In this case the order of execution is determined by a set of rules of advice precedence specified by the underlying language.
▶ There are two cases to consider:
  1. Precedence rules among advices within the same aspect.
  2. Precedence rules among advices from different aspects.

Precedence rules among advices within the same aspect

▶ There are two ways to describe precedence among advices within the same aspect. One way is to answer “[In the case of two like advices] which one executes first?” in which case the answer is “The one defined first executes first.”
▶ Another way to describe this is asking a slightly different question: “Which advice has precedence?” To answer the question we first must define “precedence.”
Precedence rules among advices within the same aspect

- In the case of two or more before advices, “precedence” has the meaning of executing first.
- In the case of after advice, “precedence” has the meaning of executing last.
- In the case of around advice, the one defined first has priority.
- Thus, to answer the question in terms of precedence, in the case of two or more before advices, the one that appears earlier in the aspect definition has precedence over the one that appears later. Otherwise, in the case of two or more after advices, the one that appears later in the aspect definition has precedence over the one that appears earlier.

Precedence rules among advices from different aspects

- An aspect definition can include an explicit declaration of precedence over another with the following statement:
  declare precedence:
  type pattern₁, type pattern₂, ..., type patternₙ
  where a type pattern must resolve to an aspect or it may include wildcard characters that must resolve to a set of aspects.
- In the above, all advices defined in an aspect (or a set of aspects) that match type pattern₁ have precedence over all advices defined in an aspect (or a set of aspects) that match type pattern₂, etc.
Precedence rules among advices from different aspects

Without an explicit declaration of precedence, if aspect Child is a subaspect of aspect Parent, then all advices defined in Child have precedence over all advices defined in Parent.

Without an explicit declaration of precedence or a super-subaspect relationship, if two pieces of advice are defined in two different aspects, precedence is undefined.

Introductions

The mechanism of introduction allows for crosscutting features (state and behavior) to be defined as part of class definitions from within aspect definitions.

It also allows one to define a given type (class or interface) as a supertype to a given type, thus modifying the class hierarchy.
Introducing behavior: An example

Consider classes Human and Noble below:

```java
public abstract class Human {
    String name;
    public Human(String name) {
        this.name = name;
    }
    public void speak() {
        System.out.println("Good morning m’lord.");
    }
}

public class Noble extends Human {
    String house;
    public Noble(String name, String house) {
        super(name);
        this.house = house;
    }
    public String toString() {
        return "I am " + this.name + ".";
    }
}
```

Introducing behavior: An example /cont.

Consider the test program below:

```java
public class Test {
    public static void main(String[] args) {
        Noble Arya = new Noble("Arya Stark", "Stark");
        Arya.speak();
    }
}
```

The output is Good morning m’lord.
Introducing behavior: An example /cont.

- The aspect below adds an overriding method speak() to class Noble.

```java
public aspect Behavior {
    public void Noble.speak() {
        System.out.println("Good morning my lord. " +
            this.toString());
    }
}
```

- The output of the program is now

  Good morning my lord. I am Arya Stark.

Introducing behavior through an interface implementation:
An example

- AspectJ allows us to declare that a class implements a given interface and thus being able to introduce behavior.

- We will extend the previous example to demonstrate this.
  Consider interface Allegiance:

```java
public interface Allegiance {
    public void declare();
}
```

- We can declare that class Noble implements Allegiance as follows:

  declare parents: Noble implements Allegiance;
Introducing behavior through an interface implementation: An example /cont.

Once we make such a declaration, we must subsequently define method `declare()`:

```java
public void Noble.declare() {
    System.out.println(this.toString() +
                        " Of House " +
                        this.house + "." );}
```

The complete aspect definition is shown below:

```java
public aspect Behavior {
    declare parents: Noble implements Allegiance;
    public void Noble.declare() {
        System.out.println(this.toString() +
                            " Of House " +
                            this.house + "." );}
    public void Noble.speak() {
        System.out.println("Good morning my lord. " +
                            this.toString());}}
```
Introducing behavior through an interface implementation: An example /cont.

Consider the test program below:

```java
public class Test {
    public static void main(String[] args) {
        Noble Arya = new Noble("Arya Stark", "Stark");
        Arya.declare();
    }
}
```

The output is I am Arya Stark. Of House Stark.

Context passing

The mechanism of context passing allows a pointcut to expose a binding to the underlying object, thus making the object available to any advice that may need to access it.

What do we mean by “underlying object”?

This would be the object where an event of interest occurs.

For example, in the case of a call join point we may be interested in the caller, the callee, or both.
Context passing: Self and target join points

- Self and target join points can capture the caller and receiver of a call.
- The join point `this` captures the sender object (caller), whereas `target` captures the receiving object (callee).
- For method executions, both join points capture the executing object.

```java
public class Point {
    protected double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0);
    }
    public void move(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "x: " + x + " y: " + y;
    }
}
```
We want to keep track of each move of each point object. In other words, we must be able to distinguish between moves per instance.

The statement

```java
int Point.numberOfMoves;
```

introduces a private integer variable `numberOfMoves` in the class definition of `Point`.

This implies that every instance of `Point` will maintain its own unique integer variable `numberOfMoves`.

Additionally, we want to be able to obtain the value of this variable.

The following definition

```java
public int Point.howMany() {
    return this.numberOfMoves;
}
```

introduces a method into class `Point` that will return the value of variable `numberOfMoves`.
Combining introductions and context passing /cont.

- We must capture the executing object upon reception of a `move()` message.
- Once we have the executing object, we can then access its own unique variable `numberOfMoves`.
- We can capture the receiving object through context passing.

  ```java
  pointcut counts(Point p) :
  
  execution(void Point.move(double, double)) &&
  this(p);
  ```

Combining introductions and context passing /cont.

- It is important to stress that pointcut `counts` performs two different tasks here:

  ```java
  pointcut counts(Point p) :
  
  execution(void Point.move(double, double)) &&
  this(p);
  ```

- It captures execution of method `move()`. This will cause any associate advice to execute.
- It captures and exposes a binding to the `Point` instance whose `move()` method is about to execute. This will allow any associated advice to obtain access to the particular object.
We define an advice to increment the variable `numberOfMoves` as follows:

```java
after(Point p) : counts(p) {
    p.numberOfMoves++;  
}
```

We can now put everything together in one aspect definition as follows:

```java
public aspect Logger {
    int Point.numberOfMoves;
    public int Point.howMany() {
        return this.numberOfMoves;
    }
    pointcut counts(Point p) :
        execution(void Point.move(double, double)) &&
        this(p);
    after(Point p) : counts(p) {
        p.numberOfMoves++;  
    }
}
```
Consider the following test program:

```java
Point p1 = new Point();
Point p2 = new Point();
p1.move(3, 7);
p1.move(3, 11);
p2.move(10, 10);
System.out.println(p1.howMany());
System.out.println(p2.howMany());
```

The output is:

2
1

Privileged aspects

- AspectJ allows us to get access to private features of a class.
- How does this compare to the friend construct in C++?
Combining context passing and privileged aspect behavior: An example

Consider the following class:

```java
public class Semaphore {
    private int value;
    public void increment() {
        this.value++;
    }
    public void decrement() {
        this.value--;
    }
    public int getValue() {
        return this.value;
    }
    public void reset() {
        this.value = 0;
    }
}
```

We want to impose a binary protocol to class Semaphore.

This means that we initially must monitor methods `increment()` and `decrement()`.

The two pointcuts capture the execution of each method respectively and expose a binding to the underlying semaphore object:

```java
pointcut monitoringIncs (Semaphore s):
    execution(* Semaphore.increment()) &&
    this(s);

pointcut monitoringDecs (Semaphore s):
    execution(* Semaphore.decrement()) &&
    this(s);
```
Combining context passing and privileged aspect behavior: An example /cont.

- Each pointcut will be associated with an advice.
- The advice for `monitoringIncs` executes instead of the code associated with the pointcut and performs a check on the value of the semaphore.
- If it is already 1, then the advice will do nothing. If it is not 1, then the advice will pass execution to the code associated with the pointcut, therefore allowing the increment.

```java
void around (Semaphore s): monitoringIncs(s) {
    if (s.value == 1)
        ;
    else
        proceed(s);
}
```

Combining context passing and privileged aspect behavior: An example /cont.

- The advice for `monitoringDecs` executes instead of the code associated with the pointcut and performs a check on the value of the semaphore.
- If it is already 0, then the advice will do nothing. If it is not 0, then the advice will pass execution to the code associated with the pointcut, therefore allowing the decrement.

```java
void around (Semaphore s): monitoringDecs(s) {
    if (s.value == 0)
        ;
    else
        proceed(s);
}
```
Combining context passing and privileged aspect behavior: An example /cont.

- Putting everything together we have the following aspect definition:

```java
public privileged aspect BinaryProtocol {
    pointcut monitoringIncs (Semaphore s):
        execution(* Semaphore.increment()) &&
        this(s);
    pointcut monitoringDecs (Semaphore s):
        execution(* Semaphore.decrement()) &&
        this(s);
    void around(Semaphore s): monitoringIncs(s) {
        if (s.value == 1)
            ;
        else
            proceed(s);}
    void around (Semaphore s): monitoringDecs(s) {
        if (s.value == 0)
            ;
        else
            proceed(s);}
}
```

Combining context passing and privileged aspect behavior: An example /cont.

- Consider the following test program:

```java
public class Test {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore();
        semaphore.increment();
        semaphore.increment();
        semaphore.decrement();
        System.out.println(semaphore.getValue());
    }
}
```

- The output is 0.
Combining introductions, context passing, and privileged aspect behavior: An example

▶ Consider the definition of class Counter:

```java
public class Counter {
    private int value;
    void increment() {
        this.value++;
    }
    public int getValue() {
        return value;
    }
}
```

Combining introductions, context passing, and privileged aspect behavior: An example /cont.

▶ We want to add cyclic behavior to counter objects, i.e. once the value of a counter object reaches some predefined maximum value, the object should reset its value. The maximum value is held by the constant

```java
private final int MAX = 10;
```

▶ Initially we define an interface that all cyclic objects must implement:

```java
public interface Cyclic {
    public void reset();
}
```
Combining introductions, context passing, and privileged aspect behavior: An example /cont.

▶ We introduce the interface implementation of class Counter together with the implementation of method reset():

declare parents: Counter implements Cyclic;

public void Counter.reset() {
    this.value = 0;
}

▶ Finally we need to be able to capture all calls to method increment() and check if variable value has reached MAX, in which case we must reset the counter. Otherwise, we allow the call to proceed.

void around(Counter c): call(* Counter.increment()) && target(c) {
    if (c.value == MAX)
        c.reset();
    proceed(c);
}

Aspect CyclicProtocol is declared privileged as it would need to obtain access to private variable value in class Counter.

public privileged aspect CyclicProtocol {
    private final int MAX = 10;
    declare parents: Counter implements Cyclic;
    public void Counter.reset() {
        this.value = 0;
    }

    void around(Counter c): call(* Counter.increment()) && target(c) {
        if (c.value == MAX)
            c.reset();
        proceed(c);
    }
}
Combining introductions, context passing, and privileged aspect behavior: An example /cont.

▶ We define a test program as follows:

```java
public class Test {
    public static void main(String[] args) {
        Counter c = new Counter();
        for (int i = 0; i < 15; i++) {
            c.increment();
            System.out.print(c.getValue() + " ");
        }
    }
}
```

▶ The output of the program is

```
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5
```

Combining introductions, context passing, and privileged aspect behavior: An example /cont.

▶ We will extend the binary semaphore example by adding locking behavior to a semaphore object.

▶ Consider interface Lockable:

```java
public interface Lockable {
    void lock();
    void unlock();
    boolean isLocked();
}
```

▶ The statement

```java
declare parents: Semaphore implements Lockable;
```

introduces interface Lockable as a supertype to class Semaphore.
The statement

```java
private boolean Semaphore.lock;
```

introduces a private integer variable lock as part of the state of class Semaphore.

The following segment

```java
public void Semaphore.lock() {
    this.lock = true;
}
public void Semaphore.unlock() {
    this.lock = false;
}
public boolean Semaphore.isLocked() {
    return this.lock;
}
```

introduces methods lock(), unlock(), and isLocked() as part of the behavior of class Semaphore.
To implement the locking mechanism, we need to intercept calls made to increment() or decrement() and place a condition that they should only be allowed to run provided that the semaphore is not locked.

We do this by first defining a pointcut that would capture any of the two calls and once it is captured it will expose a binding to the underlying semaphore object.

```java
pointcut monitoringMods (Semaphore s): 
    (call (* Semaphore.increment()) ||
     call (* Semaphore.decrement())) &&
    target(s);
```

An around advice executes instead of the code associated with the pointcut and performs a check on the status of the semaphore, only allowing the code associated with the pointcut to run provided the semaphore is not locked.

```java
void around (Semaphore s): monitoringMods(s) {
    if (s.isLocked() == true)
        System.out.println("Error: Cannot set semaphore value.");
    else
        proceed(s);
}
```
Combining introductions, context passing, and privileged aspect behavior: An example /cont.

- As the aspect definition must access private state of the class Semaphore (despite the fact that this is state introduced by the aspect itself), it must be declared privileged:

```java
class Semaphore {
    private boolean lock;

    void lock() {
        lock = true;
    }

    void unlock() {
        lock = false;
    }

    boolean isLocked() {
        return lock;
    }

    pointcut monitoringMods (Semaphore s):
        (call (* Semaphore.increment()) ||
        call (* Semaphore.decrement())) &&
        target(s);

    void around (Semaphore s): monitoringMods(s) {
        if (s.isLocked() == true)
            System.out.println("Error: Cannot set semaphore value.");
        else
            proceed(s);
    }
}
```

public privileged aspect Lock {
    declare parents: Semaphore implements Lockable;
    private boolean Semaphore.lock;
    public void Semaphore.lock() {
        this.lock = true;
    }
    public void Semaphore.unlock() {
        this.lock = false;
    }
    public boolean Semaphore.isLocked() {
        return this.lock;
    }
    pointcut monitoringMods (Semaphore s):
        (call (* Semaphore.increment()) ||
        call (* Semaphore.decrement())) &&
        target(s);
    void around (Semaphore s): monitoringMods(s) {
        if (s.isLocked() == true)
            System.out.println("Error: Cannot set semaphore value.");
        else
            proceed(s);
    }
}
Combining introductions, context passing, and privileged aspect behavior: An example /cont.

Semaphore semaphore = new Semaphore();
semaphore.increment();
semaphore.lock();
semaphore.increment();
System.out.println(semaphore.getValue());
semaphore.unlock();
semaphore.increment();
semaphore.lock();
semaphore.decrement();
semaphore.unlock();
semaphore.decrement();
System.out.println(semaphore.getValue());

Error: Cannot set semaphore value.
1
Error: Cannot set semaphore value.
0

Reusing pointcuts: Abstract aspects

- Even though the adoption of AOP results in a good separation of concerns, restricting aspect definitions to match class and method names of system core concerns leads to strong binding between aspects and system core concerns.
- In such cases aspect definitions are not reusable, but they are restricted to be only applicable in one specific application context.
To be able to deploy an aspect definition in different contexts, we first need to answer the following questions: “What needs to be reused?” and “What part of an aspect definition can be bound to the core functionality?”

An obvious construct which binds an aspect to the core functionality is the (named or anonymous) pointcut.

In order to support reuse, a level of genericity is supported by AspectJ through the provision of abstract aspects. We can distinguish between two cases, discussed in the subsequent subsections.

Reusing concrete pointcuts

A concrete pointcut expression can be reused not only by advices within the aspect where it is being defined, but by advices in all subaspects.

One restriction imposed by AspectJ is that in order to define a subaspect, the superaspect must itself be declared abstract (even in the case where the superaspect contains no abstract feature).
Reusing concrete pointcuts /cont.

- Much like class features, pointcut declarations can be associated with access modifiers:
  - `public`: The declaration can be visible to all aspects anywhere in the application.
  - No modifier: This is the default. It implies that the declaration is visible to all aspects within the same package.
  - `protected`: The declaration is visible to host aspect and all its subaspects.
  - `private`: The declaration is visible only to the host aspect.

Reusing abstract pointcuts

- A pointcut can be declared abstract when we do not want to commit to a particular application in the current aspect definition but we prefer to leave the concrete definition in subaspects.
- This idea allows the development of aspect libraries, as collections of generic aspect definitions.
- Much like a concrete subclass which inherits from an abstract superclass must implement all inherited abstract methods or must itself be declared abstract, a subaspect must provide a definition of all abstract pointcuts inherited from an abstract supersaspect, otherwise it must itself be declared abstract.
Reusing abstract pointcuts: An example

- We want to build an aspect that will implement a generic tracing facility.
- The abstract aspect AbstractLogger does not implement a pointcut, but it does provide a reflection-based tracing facility upon entering and exiting the code to be defined by some concrete pointcut in a subaspect.

```java
public abstract aspect AbstractLogger {
    abstract pointcut monitored();
    before(): monitored() {
        System.out.println(">Entering: " + thisJoinPoint);
    }
    after(): monitored() {
        System.out.println(">Exiting: " + thisJoinPoint);
    }
}
```

Reusing abstract pointcuts: An example /cont.

- Aspect ConcreteLogger inherits from AbstractLogger and implements the abstract pointcut monitored().

```java
public aspect ConcreteLogger extends AbstractLogger {
    pointcut monitored(): execution(void Stack.push(String)) ||
        execution(String Stack.pop());
}
```
Consider the following test program:

```java
Stack myStack = new Stack();
myStack.push("base");
myStack.push("your");
myStack.push("all");
System.out.println(myStack.pop());
System.out.println(myStack.pop());
System.out.println(myStack.pop());
```

The output is as follows:

```
>Entering: execution(void Stack.push(String))
>Exiting: execution(void Stack.push(String))
>Entering: execution(void Stack.push(String))
>Exiting: execution(void Stack.push(String))
>Entering: execution(void Stack.push(String))
>Exiting: execution(void Stack.push(String))
>Entering: execution(void Stack.push(String))
>Exiting: execution(void Stack.push(String))
>Entering: execution(String Stack.pop())
>Exiting: execution(String Stack.pop())
 all
>Entering: execution(String Stack.pop())
>Exiting: execution(String Stack.pop())
 your
>Entering: execution(String Stack.pop())
>Exiting: execution(String Stack.pop())
 base
```