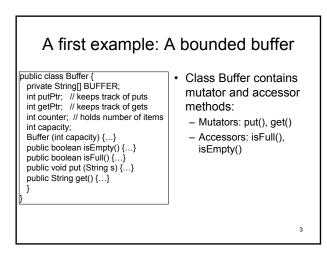
#### An introduction to Aspect-Oriented Programming with AspectJ

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.



#### 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++: 3 } public String get() { if (isEmpty()) return "ERROR: Buffer empty"; else { counterreturn 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. <u>Pointcut</u>: A collection of joinpoints. – e.g. the evection of all mutator methods inside class

- e.g. the execution of all mutator methods inside class Buffer.
- <u>Advice</u>: 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.

5

#### 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())

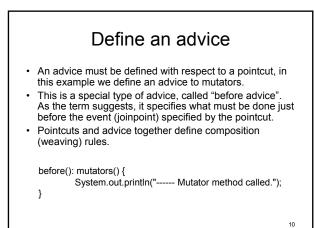
#### Defining a pointcut

• 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

- 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

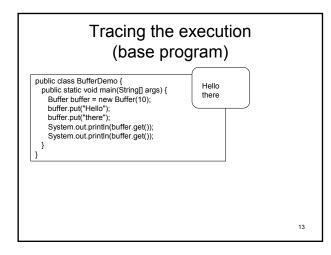


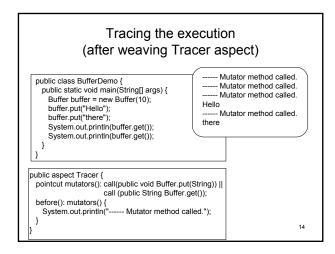
## An advice associates the code to be executed with pointcuts. There are three ways to associate an advice with a pointcut: Before: run just before the pointcut.

- 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.





## Types of joinpointsPatter1. Calls to methods and constructors• Pointcuts use<br/>specify event2. Execution of methods and constructors• Pointcuts use<br/>specify event3. Field access• The use of the<br/>overloaded.4. Exception handling• Using \* when<br/>any type.5. Class initialization- Using \* when<br/>any type.6. Lexical structure- Using \* when<br/>any type.7. Control flow- Using \* when<br/>matches any<br/>You can place

9. Conditional test

15

## 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

#### 21

19

### 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]

22

20

call (public static void	Call to public static myMethod() in
MyClass.myMethod(String))	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.	
	25	

<ul> <li>Capture read and write</li> <li>The general format is get (FieldSignature) or set (</li> <li>FieldSignature is</li> </ul>	access to the fields of a class. FieldSignature)
[modifier_pattern] ty	/pe_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 > of type int in MyClass.

Exception handling		
<ul> <li>Capture the execution of exception handlers of specified types.</li> </ul>		
<ul> <li>The general form is</li> </ul>		
handler(ExceptionTypePattern)		
i i ( i i più ) più i i i i i i i i i i i i i i i i i i		
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". 27	

## <text><list-item><list-item>

Lexical structure		
<ul> <li>Capture joinpoints inside t a method.</li> </ul>	he lexical structure of class or	
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. 29	

Control flow		
<ul> <li>Capture joinpoints based on the control flow of other joinpoints.</li> <li>e.g. if a() calls b(), then b() is within the control flow of a().</li> <li>Take the forms cflow (joinpoint) cflowbelow(joinpoint)</li> </ul>		
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. 30	

#### 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 31

#### 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

# <text><list-item><list-item><text>

#### 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)

#### 34

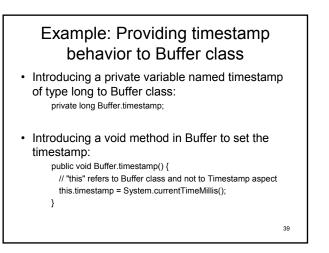
### Example: Tracing with reflection Let us trace the execution of all methods inside class Buffer, with any type of arguments, returning any type and with any access type. The pointcut below specifies the above events: execution (\* Buffer.\*(..)) This is an example of a named pointcut: pointcut publics():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

<section-header>A Cracing aspect with reflectionpublic aspect ReflectionTracer {<br/>pointcut publics(): execution (\* Buffer.\*(..));<br/>before(): publics() {<br/>grefner(): publics() {<br/>gr

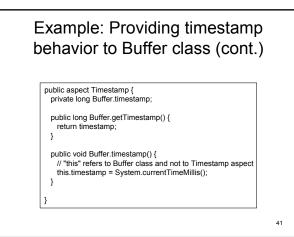
Running the tracir	
<pre>public class BufferDemo {     public static void main(String[] args) {         Buffer buffer = new Buffer(10);         buffer.put("Hello");         buffer.put("there");         System.out.println(buffer.get());         System.out.println(buffer.get());     } }</pre>	Before: execution(void 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)) After: execution(boolean Buffer,isFull()) After: execution(boolean Buffer,isFull()) After: execution(boolean Buffer,isFull()) Before: execution(String Buffer,get()) Before: execution(boolean Buffer,isEmpty()) After: execution(boolean Buffer,isEmpty()) After: execution(boolean Buffer,isEmpty()) After: execution(boolean Buffer,isEmpty())
public aspect ReflectionTracer { pointcut publics(): execution (* Buffer.*()); before(): publics() { System.out.println("Before: " + thisJoinPoint); } affer(): publics() { System.out.println("After: " + thisJoinPoint);	Before: execution(String Buffer.get()) Before: execution(bolean Buffer isEmpty()) After: execution(bolean Buffer isEmpty()) After: execution(String Buffer.get()) there
}	37

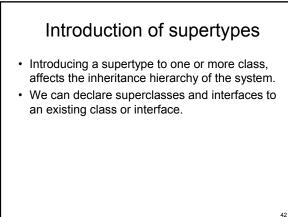
#### Type modification constructs

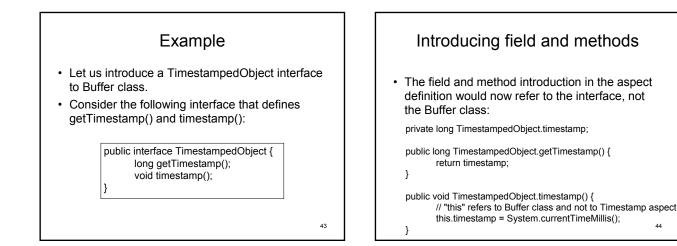
- Using an advice we are able to affect the dynamic behavior of a system.
- · Sometimes it is necessary to provide aspectual behavior over the static structure of the system.
- · AspectJ allows a number of staticcrosscutting types, including:
  - Introduction of new methods and fields.
  - Introduction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

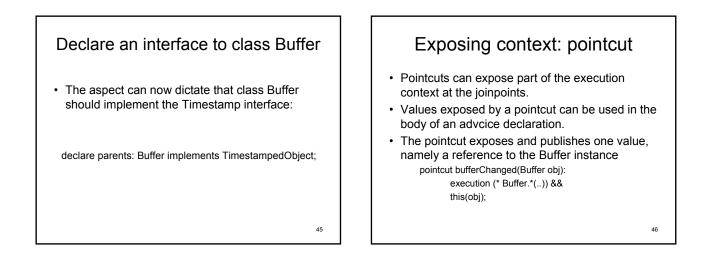


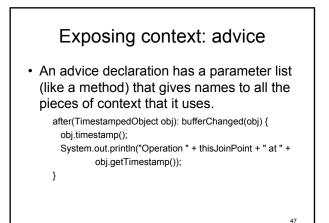
#### Example: Providing timestamp behavior to Buffer class (cont.) · Introducing a long method in Buffer to return the timestamp: public long Buffer.getTimestamp() { return timestamp; } 40

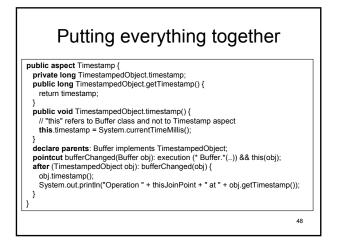












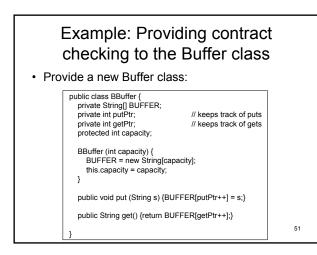
#### Operation execution(boolean Buffer.isFull()) at 1096152607327 Operation execution(void Buffer.put(String)) at 1096152607327 Operation execution(void Buffer.put(String)) at 1096152607327 Operation execution(boolean Buffer.isEmpty()) at 1096152607327 Operation execution(boolean Buffer.isEmpty()) at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

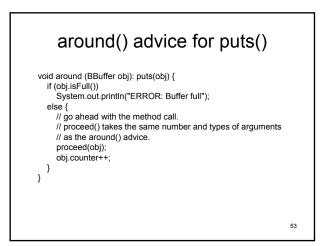
49

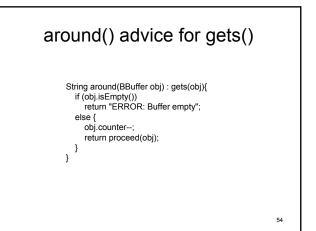
#### Around advice

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

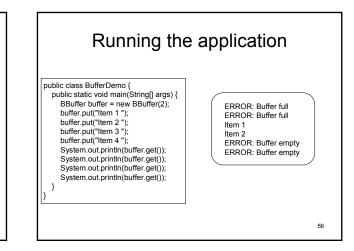
50

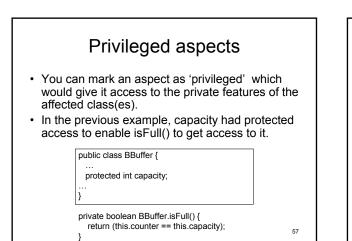


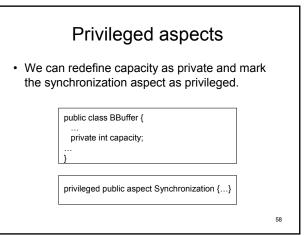




#### Synchronization aspect public aspect Synchronization { private int BBuffer.counter = 0; private boolean BBuffer.isEmpty() {return (this.counter==0);} private boolean BBuffer.isFull() {return (this.counter == this.capacity);} pointcut puts(BBuffer obj): execution (\* BBuffer.put(String)) && this(obj); 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 { obi.counter--: 55 return proceed(obj); }}}





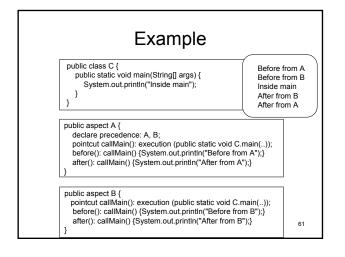


## Determining precedence among advice

- 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

- 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.
- 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.
- Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.



### 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.

