Advanced Object-Oriented Design
Lecture 14

Aspect-Oriented Programming

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Agenda

Aspect Oriented Programming

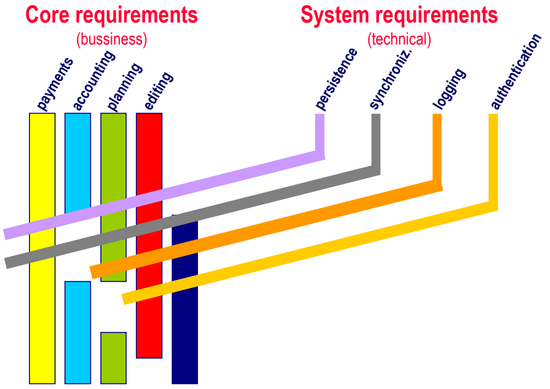
Objects are not enough

Object-oriented paradigm

- system is a set of collaborating units – classes
- classes allow for hiding their implementation and expose interfaces only
- polymorphism localizes common behavior and provides interface for related concepts, and allows for specializations

but OO paradigm fails to address properly functions that span multiple unrelated units

Requirements decomposition



Implications

- **Code tangling**
  Modules in a software system may simultaneously interact with several requirements. For example, often developers simultaneously think about business logic, performance, synchronization, logging, and security. Such a multitude of requirements results in the simultaneous presence of elements from each concern's implementation, resulting in code tangling.
- **Code scattering**
  Since crosscutting concerns, by definition, spread over many modules, related implementations also spread over all those modules. For example, in a system using a database, performance concerns may affect all the modules accessing the database.

http://www.javaworld.com/

Concerns

A ***concern*** is a particular goal, concept, or area of interest.

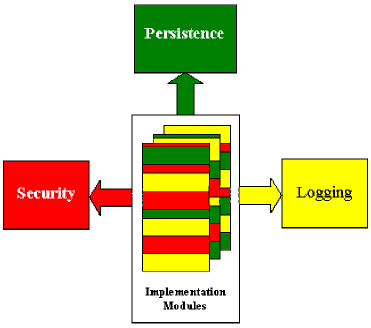
In technology terms, a typical software system comprises several core and system-level concerns.

For example, a credit card processing system's *core concern* would process payments, while its *system-level concerns* would handle logging, transaction integrity, authentication, security, performance, and so on. Many such concerns – known as *crosscutting concerns* – tend to affect multiple implementation modules.

Using current programming methodologies, crosscutting concerns span over multiple modules, resulting in systems that are harder to design, understand, implement, and evolve.

AspectJ homepage

System = a set of concerns



http://www.javaworld.com/

Agenda

AspectJ basics

PARC AspectJ

- a freely available AOP implementation for Java from Xerox PARC
- uses Java as the language for implementing individual concerns,
- specifies extensions to Java for weaving rules
- the rules are specified in terms of *pointcuts, join points, advices,* and *aspects*
- aspect compiler combines different aspects and the original code together at byte-code level

Xerox Palo-Alto Research Center

Hello, World!

```
public class HelloWorld {
   public static void say(String message) {   \
      System.out.println(message);
   }
   public static void sayToPerson(String message, String name) {
      System.out.println(name + ", " + message);
   }
}
```

AspectJ homepage

Hello, World! (cont.)

```
public aspect MannersAspect {
   pointcut callSayMessage() :
      call(public static void HelloWorld.say*(..));

   before() : callSayMessage() {
      System.out.println("Good day!");
   }

   after() : callSayMessage() {
      System.out.println("Thank you!");
   }
}
```

AspectJ homepage

Basic AspectJ concepts

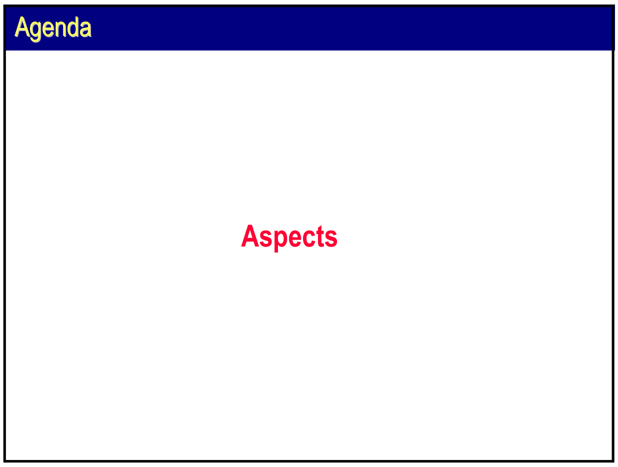
- **joinpoint**

  an identifiable point in a program's execution; e.g. joinpoints could define calls to specific methods in a class, loops, assignments, handling exceptions etc.
- **pointcut**

  program construct to designate joinpoints and collect specific context at those points
- **advice**

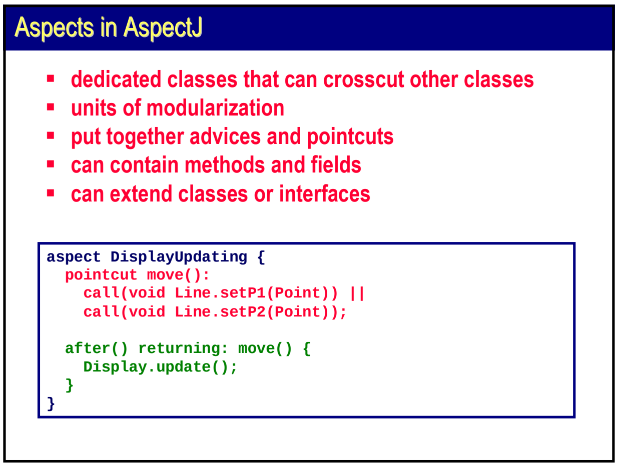
  code that runs upon meeting certain conditions: before, after and around a pointcut; e.g. an advice could log a message before executing a joinpoint

AspectJ homepage



Agenda

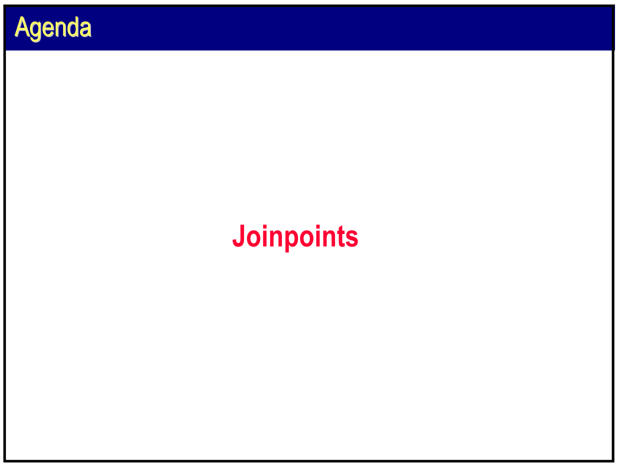
Aspects



Aspects in AspectJ

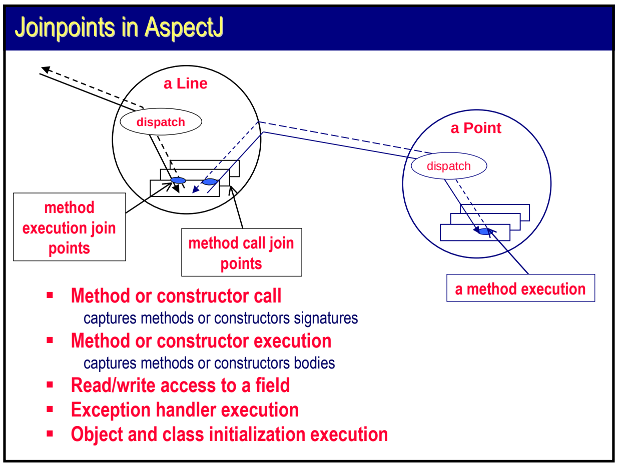
- dedicated classes that can crosscut other classes
- units of modularization
- put together advices and pointcuts
- can contain methods and fields
- can extend classes or interfaces

```
aspect DisplayUpdating {
  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() returning: move() {
    Display.update();
  }
}
```



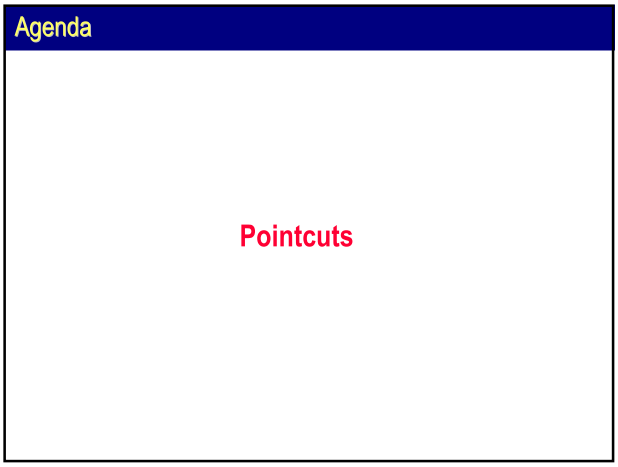
Agenda

Joinpoints



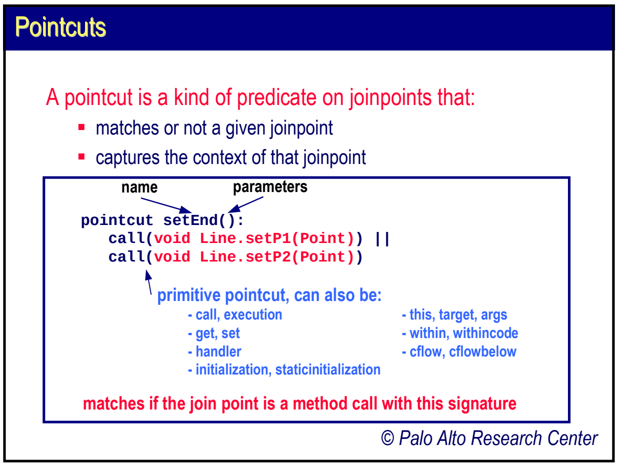
Joinpoints in AspectJ

- Method or constructor call
  captures methods or constructors signatures
- Method or constructor execution
  captures methods or constructors bodies
- Read/write access to a field
- Exception handler execution
- Object and class initialization execution



Agenda

Pointcuts



Pointcuts

A pointcut is a kind of predicate on joinpoints that:

- matches or not a given joinpoint
- captures the context of that joinpoint

name
parameters

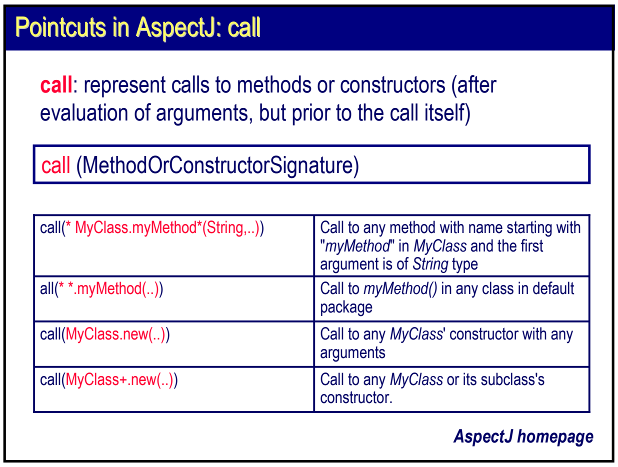
```
pointcut setEnd():
   call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point))
```

primitive pointcut, can also be:

- call, execution
- get, set
- handler
- initialization, staticinitialization
- this, target, args
- within, withincode
- cflow, cflowbelow

matches if the join point is a method call with this signature

© Palo Alto Research Center



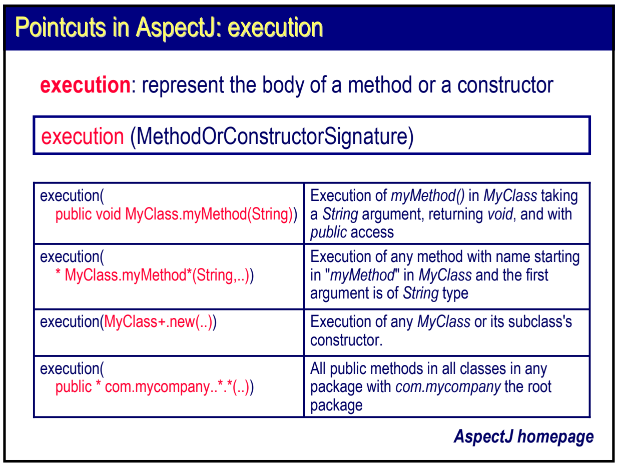
Pointcuts in AspectJ: call

call: represent calls to methods or constructors (after evaluation of arguments, but prior to the call itself)

call (MethodOrConstructorSignature)

call(* MyClass.myMethod*(String,..))	Call to any method with name starting with "*myMethod*" in *MyClass* and the first argument is of *String* type
all(* *.myMethod(..))	Call to *myMethod()* in any class in default package
call(MyClass.new(..))	Call to any *MyClass*' constructor with any arguments
call(MyClass+.new(..))	Call to any *MyClass* or its subclass's constructor.

AspectJ homepage



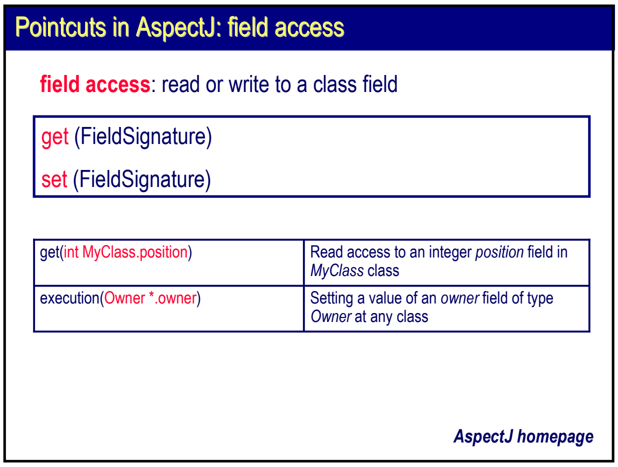
Pointcuts in AspectJ: execution

execution: represent the body of a method or a constructor

execution (MethodOrConstructorSignature)

execution(public void MyClass.myMethod(String))	Execution of *myMethod()* in *MyClass* taking a *String* argument, returning *void*, and with *public* access
execution(* MyClass.myMethod*(String,..))	Execution of any method with name starting in "*myMethod*" in *MyClass* and the first argument is of *String* type
execution(MyClass+.new(..))	Execution of any *MyClass* or its subclass's constructor.
execution(public * com.mycompany..*.*(..))	All public methods in all classes in any package with *com.mycompany* the root package

AspectJ homepage



Pointcuts in AspectJ: field access

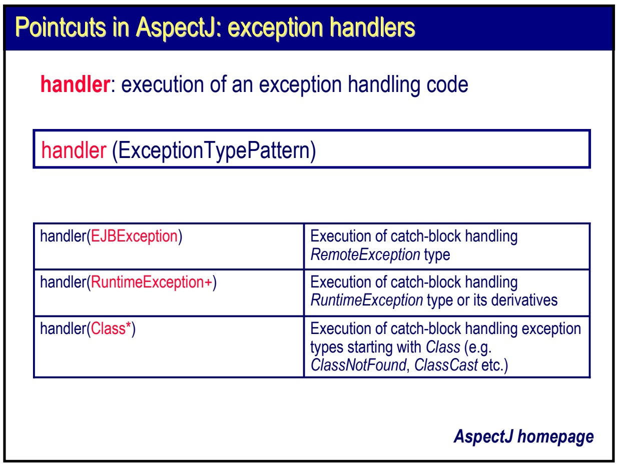
field access: read or write to a class field

get (FieldSignature)

set (FieldSignature)

get(int MyClass.position)	Read access to an integer *position* field in *MyClass* class
execution(Owner *.owner)	Setting a value of an *owner* field of type *Owner* at any class

AspectJ homepage



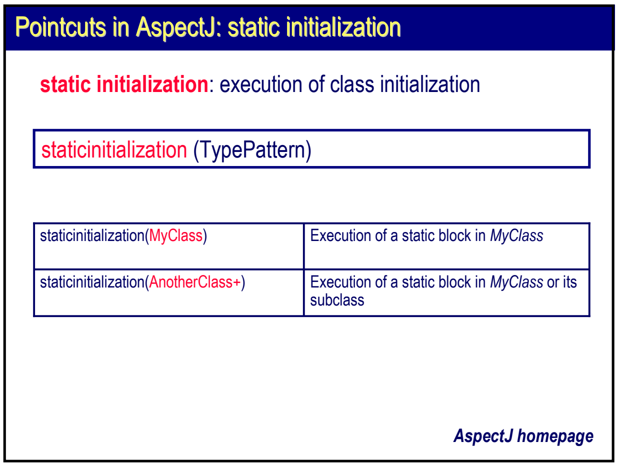
Pointcuts in AspectJ: exception handlers

handler: execution of an exception handling code

handler (ExceptionTypePattern)

handler(EJBException)	Execution of catch-block handling *RemoteException* type
handler(RuntimeException+)	Execution of catch-block handling *RuntimeException* type or its derivatives
handler(Class*)	Execution of catch-block handling exception types starting with *Class* (e.g. *ClassNotFound*, *ClassCast* etc.)

AspectJ homepage



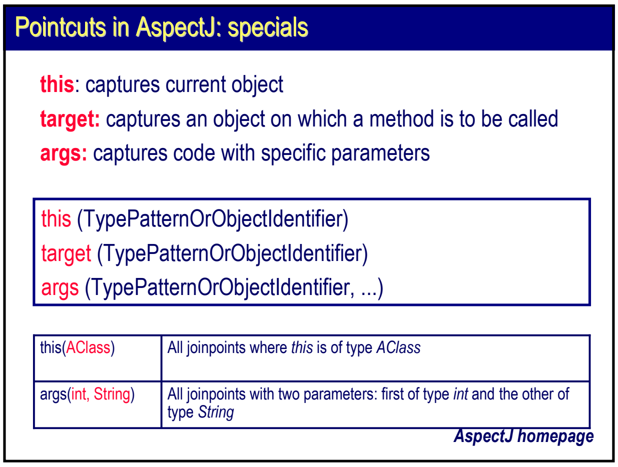
Pointcuts in AspectJ: static initialization

static initialization: execution of class initialization

staticinitialization (TypePattern)

staticinitialization(MyClass)	Execution of a static block in *MyClass*
staticinitialization(AnotherClass+)	Execution of a static block in *MyClass* or its subclass

AspectJ homepage



Pointcuts in AspectJ: specials

this: captures current object

target: captures an object on which a method is to be called

args: captures code with specific parameters

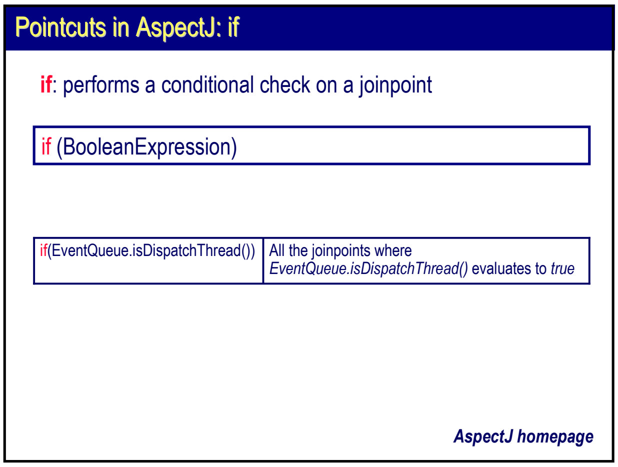
this (TypePatternOrObjectIdentifier)

target (TypePatternOrObjectIdentifier)

args (TypePatternOrObjectIdentifier, ...)

this(AClass)	All joinpoints where *this* is of type *AClass*
args(int, String)	All joinpoints with two parameters: first of type *int* and the other of type *String*

AspectJ homepage



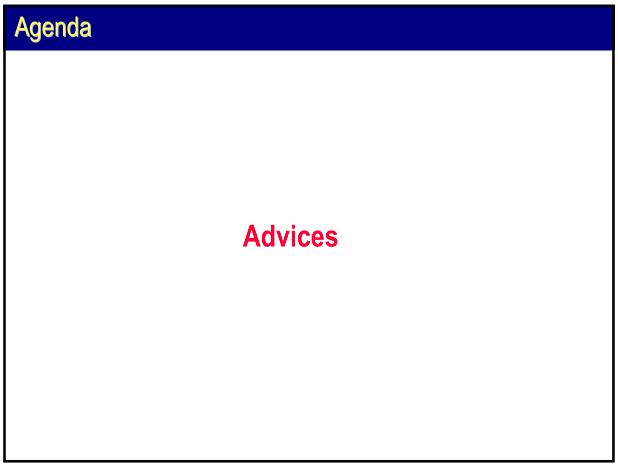
Pointcuts in AspectJ: if

if: performs a conditional check on a joinpoint

if (BooleanExpression)

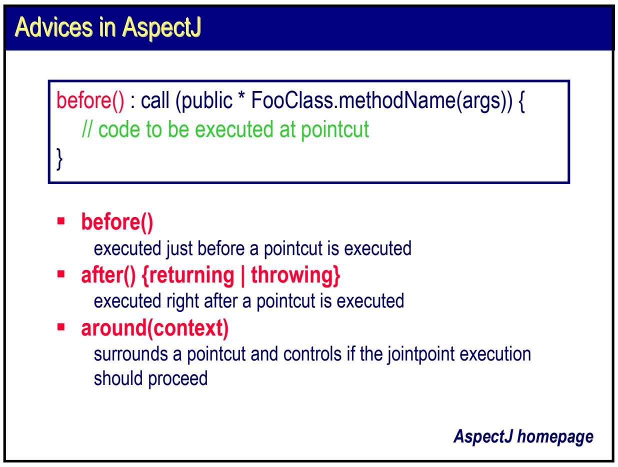
if(EventQueue.isDispatchThread())	All the joinpoints where *EventQueue.isDispatchThread()* evaluates to *true*

AspectJ homepage



Agenda

Advices

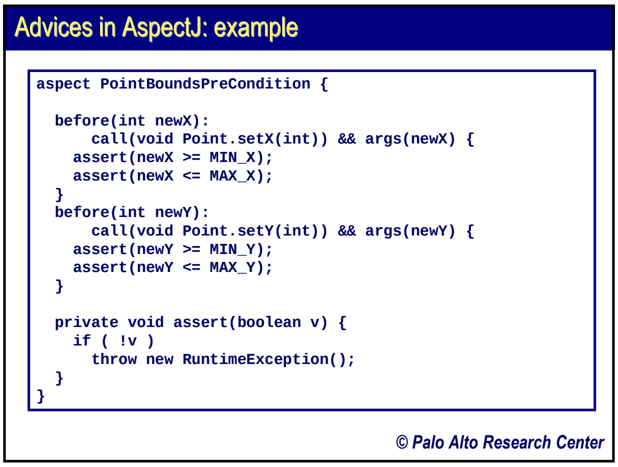


Advices in AspectJ

```
before() : call (public * FooClass.methodName(args)) {
    // code to be executed at pointcut
}
```

- **before()**
  executed just before a pointcut is executed
- **after() {returning | throwing}**
  executed right after a pointcut is executed
- **around(context)**
  surrounds a pointcut and controls if the jointpoint execution should proceed

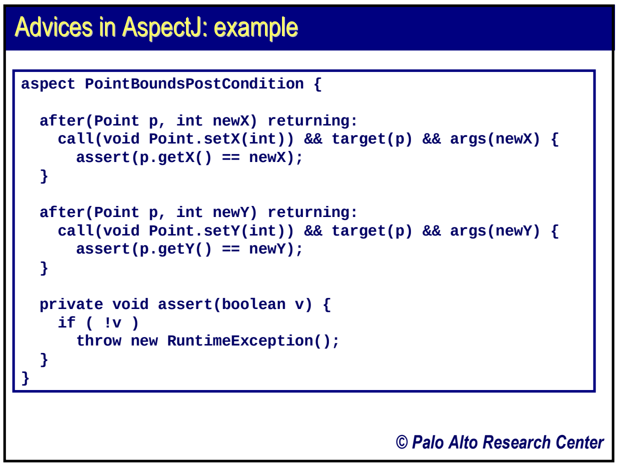
AspectJ homepage



Advices in AspectJ: example

```
aspect PointBoundsPreCondition {

  before(int newX):
      call(void Point.setX(int)) && args(newX) {
    assert(newX >= MIN_X);
    assert(newX <= MAX_X);
  }
  before(int newY):
      call(void Point.setY(int)) && args(newY) {
    assert(newY >= MIN_Y);
    assert(newY <= MAX_Y);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```

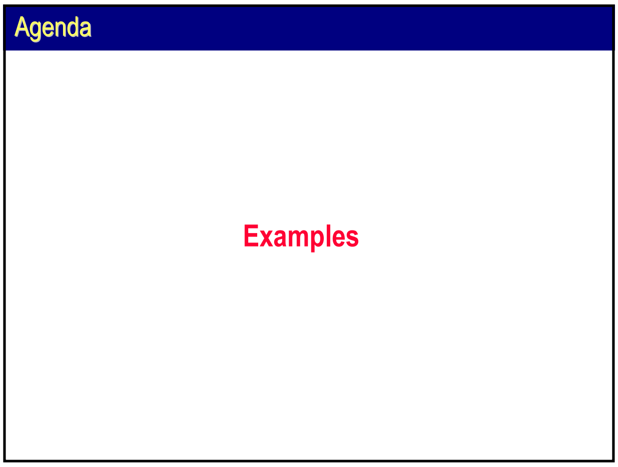
© Palo Alto Research Center



Advices in AspectJ: example

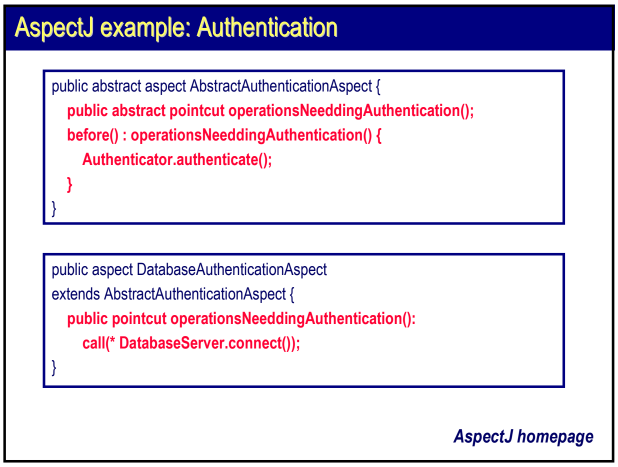
```
aspect PointBoundsPostCondition {

  after(Point p, int newX) returning:
    call(void Point.setX(int)) && target(p) && args(newX) {
      assert(p.getX() == newX);
  }

  after(Point p, int newY) returning:
    call(void Point.setY(int)) && target(p) && args(newY) {
      assert(p.getY() == newY);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```

© Palo Alto Research Center



Agenda

Examples

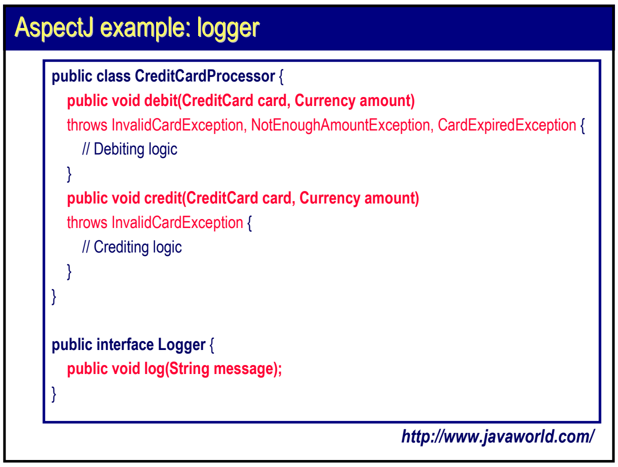


AspectJ example: Authentication

```
public abstract aspect AbstractAuthenticationAspect {
    public abstract pointcut operationsNeeddingAuthentication();
    before() : operationsNeeddingAuthentication() {
        Authenticator.authenticate();
    }
}
```

```
public aspect DatabaseAuthenticationAspect
extends AbstractAuthenticationAspect {
    public pointcut operationsNeeddingAuthentication():
        call(* DatabaseServer.connect());
}
```

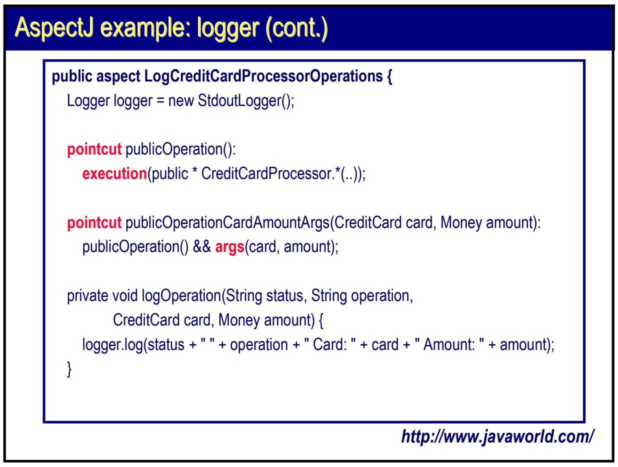
AspectJ homepage



AspectJ example: logger

```
public class CreditCardProcessor {
    public void debit(CreditCard card, Currency amount)
    throws InvalidCardException, NotEnoughAmountException, CardExpiredException {
        // Debiting logic
    }
    public void credit(CreditCard card, Currency amount)
    throws InvalidCardException {
        // Crediting logic
    }
}

public interface Logger {
    public void log(String message);
}
```

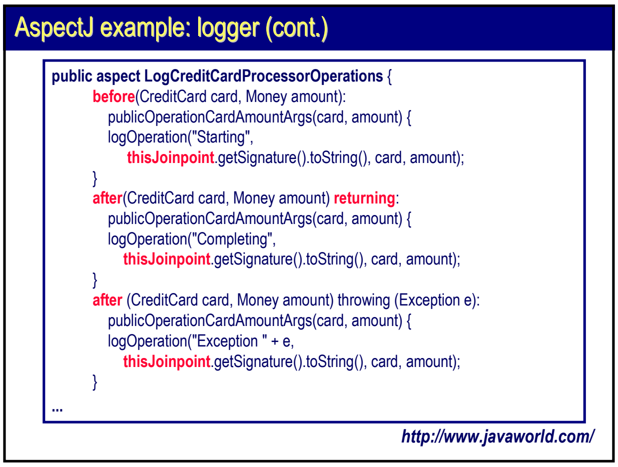
http://www.javaworld.com/



AspectJ example: logger (cont.)

```
public aspect LogCreditCardProcessorOperations {
    Logger logger = new StdoutLogger();

    pointcut publicOperation():
        execution(public * CreditCardProcessor.*(..));

    pointcut publicOperationCardAmountArgs(CreditCard card, Money amount):
        publicOperation() && args(card, amount);

    private void logOperation(String status, String operation,
              CreditCard card, Money amount) {
        logger.log(status + " " + operation + " Card: " + card + " Amount: " + amount);
    }
```

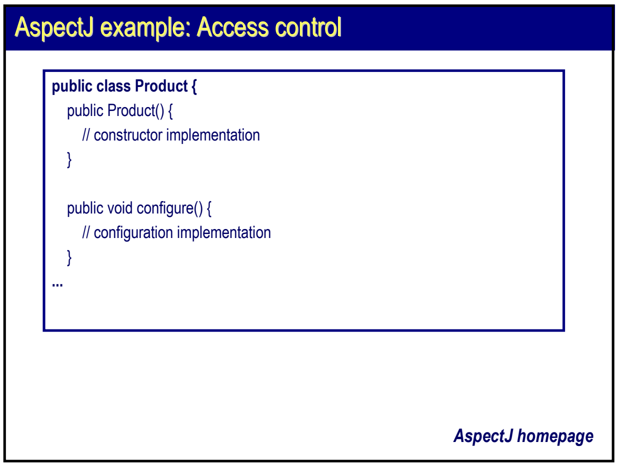
http://www.javaworld.com/



AspectJ example: logger (cont.)

```
public aspect LogCreditCardProcessorOperations {
          before(CreditCard card, Money amount):
             publicOperationCardAmountArgs(card, amount) {
             logOperation("Starting",
                 thisJoinpoint.getSignature().toString(), card, amount);
          }
          after(CreditCard card, Money amount) returning:
             publicOperationCardAmountArgs(card, amount) {
             logOperation("Completing",
                thisJoinpoint.getSignature().toString(), card, amount);
          }
          after (CreditCard card, Money amount) throwing (Exception e):
             publicOperationCardAmountArgs(card, amount) {
             logOperation("Exception " + e,
                thisJoinpoint.getSignature().toString(), card, amount);
          }
...
```

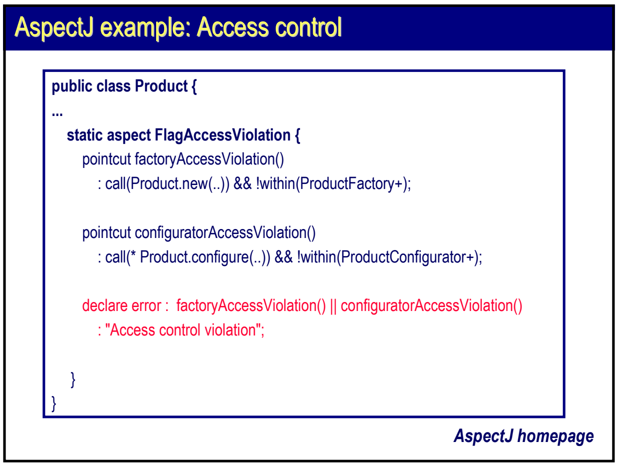
http://www.javaworld.com/



AspectJ example: Access control

```
public class Product {
    public Product() {
        // constructor implementation
    }

    public void configure() {
        // configuration implementation
    }
...
```

AspectJ homepage



AspectJ example: Access control

```
public class Product {
...
    static aspect FlagAccessViolation {
        pointcut factoryAccessViolation()
            : call(Product.new(..)) && !within(ProductFactory+);

        pointcut configuratorAccessViolation()
            : call(* Product.configure(..)) && !within(ProductConfigurator+);

        declare error :  factoryAccessViolation() || configuratorAccessViolation()
            : "Access control violation";

    }
}
```

AspectJ homepage

AspectJ example: Access control

```
public class Product {
...
   static aspect FlagAccessViolation {
      pointcut factoryAccessViolation()
         : call(Product.new(..)) && !within(ProductFactory+);

      pointcut configuratorAccessViolation()
         : call(* Product.configure(..)) && !within(ProductConfigurator+);

      before() :  factoryAccessViolation() || configuratorAccessViolation() {
         throw new Exception("Access control violation");
      }
   }
}
```

AspectJ homepage

Other capabilities

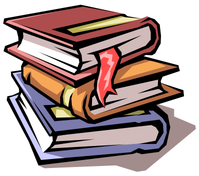
- **method and field introduction**
  introducing methods and fields into classes and interfaces
- **restructuring inheritance hierarchy**
  defining a parent class or implemented interfaces
- **translating checked exceptions to unchecked ones**
  wrapping exceptions into *org.aspectj.lang.SoftException*
- **accessing non-public members**
  priviliged aspects access private and protected class member

AspectJ homepage

Summary

- **AOP is based on, not contrary to, other programming paradigms**
- **AOP helps in modularizing the code**
- **aspects crosscut the existing modules**
- **AOP allows for altering both behavior and structure of the code**

Readings



1. ***http://eclipse.org/aspectj/***
2. ***http://www.parc.com/research/csl/projects/aspectj/default.html***
3. ***Kiczales G. et al. "An overview of AspectJ". Proceedings of 15th ECOOP***

Q & A