# The AspectJTM Programming Guide

# Table of Contents

# Table of Contents

# The AspectJ™ Programming Guide

## the AspectJ Team



**Abstract**

This programming guide describes the AspectJ language. A companion guide describes the tools which are part of the AspectJ development environment.

If you are completely new to AspectJ, you should first read Getting Started with AspectJ for a broad overview of programming in AspectJ. If you are already familiar with AspectJ, but want a deeper understanding, you should read The AspectJ Language and look at the examples in the chapter. If you want a more formal definition of AspectJ, you should read Semantics.

---

**Table of Contents**

# Preface

This user's guide does three things. It

- introduces the AspectJ language
- defines each of AspectJ's constructs and their semantics, and
- provides examples of their use.

Three appendices give a quick reference, a more formal definition of AspectJ, and a glossary.

The first section, Getting Started with AspectJ, provides a gentle overview of writing AspectJ programs. It also shows how one can introduce AspectJ into an existing development effort in stages, reducing the associated risk. You should read this section if this is your first exposure to AspectJ and you want to get a sense of what AspectJ is all about.

The second section, The AspectJ Language, covers the features of the language in more detail, using code snippets as examples. The entire language is covered, and after reading this section, you should be able to use all the features of the language correctly.

The next section, Examples, comprises a set of complete programs that not only show the features being used, but also try to illustrate recommended practice. You should read this section after you are familiar with the elements of AspectJ.

The back matter contains several appendices that cover AspectJ's semantics, a quick reference to the language, a glossary of terms and the index.

# Chapter 1. Getting Started with AspectJ

**Table of Contents**

# Introduction

Many software developers are attracted to the idea of aspect−oriented programming (AOP) but unsure about how to begin using the technology. They recognize the concept of crosscutting concerns, and know that they have had problems with the implementation of such concerns in the past. But there are many questions about how to adopt AOP into the development process. Common questions include:

- Can I use aspects in my existing code?
- What kinds of benefits can I expect to get from using aspects?
- How do I find aspects in my programs?
- How steep is the learning curve for AOP?
- What are the risks of using this new technology?

This chapter addresses these questions in the context of AspectJ a general−purpose aspect−oriented extension to Java. A series of abridged examples illustrate the kinds of aspects programmers may want to implement using AspectJ and the benefits associated with doing so. Readers who would like to understand the examples in more detail, or who want to learn how to program examples like these, can find the complete examples and supporting material on the AspectJ web site(http://aspectj.org/documentation/papersAndSlides/figures−cacm2001.zip).

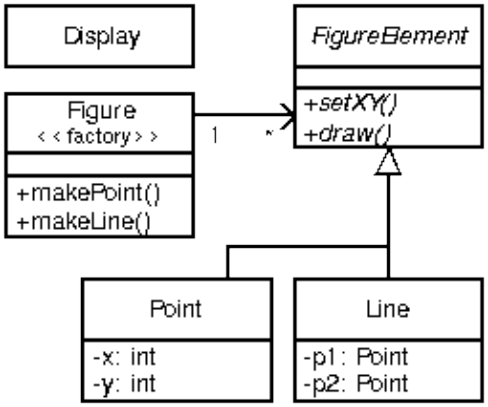
A significant risk in adopting any new technology is going too far too fast. Concern about this risk causes many organizations to be conservative about adopting new technology. To address this issue, the examples in this chapter are grouped into three broad categories, with aspects that are easier to adopt into existing development projects coming earlier in this chapter. The next section, AspectJ Semantics, we present the core of AspectJ's semantics, and in Development Aspects, we present aspects that facilitate tasks such as debugging, testing and performance tuning of applications. And, in the section following, Production Aspects, we present aspects that implement crosscutting functionality common in Java applications. We will defer discussing a third category of aspects, reusable aspects until The AspectJ Language.

These categories are informal, and this ordering is not the only way to adopt AspectJ. Some developers may want to use a production aspect right away. But our experience with current AspectJ users suggests that this is one ordering that allows developers to get experience with (and benefit from) AOP technology quickly, while also minimizing risk.

# AspectJ Semantics

This section presents a brief introduction to the features of AspectJ used later in this chapter. These features are at the core of the language, but this is by no means a complete overview of AspectJ.

The semantics are presented using a simple figure editor system. A `Figure` consists of a number of `FigureElements`, which can be either `Point`s or `Line`s. The `Figure` class provides factory services. There is also a `Display`. Most example programs later in this chapter are based on this system as well.




UML for the `FigureEditor` example

The motivation for AspectJ (and likewise for aspect−oriented programming) is the realization that there are issues or concerns that are not well captured by traditional programming methodologies. Consider the problem of enforcing a security policy in some application. By its nature, security cuts across many of the natural units of modularity of the application. Moreover, the security policy must be uniformly applied to any additions as the application evolves. And the security policy that is being applied might itself evolve. Capturing concerns like a security policy in a disciplined way is difficult and error−prone in a traditional programming language.

Concerns like security cut across the natural units of modularity. For object−oriented programming languages, the natural unit of modularity is the class. But in object−oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these they aren't reusable, they can't be refined or inherited, they are spread through out the program in an undisciplined way, in short, they are difficult to work with.

Aspect−oriented programming is a way of modularizing crosscutting concerns much like object−oriented programming is a way of modularizing common concerns. AspectJ is an implementation of aspect−oriented programming for Java.

AspectJ adds to Java just one a new concept, a join point, and a few new constructs: pointcuts, advice, introduction and aspects. Pointcuts and advice dynamically affect program flow, and introduction statically

affects a program's class heirarchy.

A *join point* is a well−defined point in the program flow. *Pointcuts* select certain join points and values at those points. *Advice* defines code that is executed when a pointcut is reached. These are, then, the dynamic parts of AspectJ.

AspectJ also has a way of affecting a program statically. *Introduction* is how AspectJ modifies a program's static structure, namely, the members of its classes and the relationship between classes.

The last new construct in AspectJ is the *aspect*. Aspects, are AspectJ's unit of modularity for crosscutting concerns They are defined in terms of pointcuts, advice and introduction.

In the sections immediately following, we are first going to look at join points and how they compose into pointcuts. Then we will look at advice, the code which is run when a pointcut is reached. We will see how to combine pointcuts and advice into aspects, AspectJ's reusable, inheritable unit of modularity. Lastly, we will look at how to modify a program's class structure with introduction.

## The Dynamic Join Point Model

A critical element in the design of any aspect−oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns.

This chapter describes AspectJ's dynamic join points, in which join points are certain well−defined points in the execution of the program. Later we will discuss introduction, AspectJ's form for modifying a program statically.

AspectJ provides for many kinds of join points, but this chapter discusses only one of them: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including normal or abrupt return.

Each method call itself is one join point. The dynamic context of a method call may include many other join points: all the join points that occur when executing the called method and any methods that it calls.

## Pointcut Designators

In AspectJ, *pointcut designators* (or simply pointcuts) identify certain join points in the program flow. For example, the pointcut

```
call(void Point.setX(int))
```

identifies any call to the method `setX` defined on `Point` objects. Pointcuts can be composed using a filter composition semantics, so for example:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

identifies any call to either the `setX` or `setY` methods defined by `Point`.

Programmers can define their own pointcuts, and pointcuts can identify join points from many different classes  in other words, they can crosscut classes. So, for example, the following declares a new, named pointcut:

```
pointcut move(): call(void FigureElement.setXY(int,int)) ||
                 call(void Point.setX(int))              ||
                 call(void Point.setY(int))              ||
                 call(void Line.setP1(Point))            ||
                 call(void Line.setP2(Point));
```

The effect of this declaration is that `move` is now a pointcut that identifies any call to methods that move figure elements.

### Property−Based Primitive Pointcuts

The previous pointcuts are all based on explicit enumeration of a set of method signatures. We call this *name−based* crosscutting. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property−based* crosscutting. The simplest of these involve using wildcards in certain fields of the method signature. For example:

```
call(void Figure.make*(..))
```

identifies calls to any method defined on `Figure`, for which the name begins with "`make`", specifically the factory methods `makePoint` and `makeLine`; and

```
call(public * Figure.* (..))
```

identifies calls to any public method defined on `Figure`.

One very powerful primitive pointcut, `cflow`, identifies join points based on whether they occur in the dynamic context of another pointcut. So

```
cflow(move())
```

identifies all join points that occur between receiving method calls for the methods in `move` and returning from those calls (either normally or by throwing an exception.)

## Advice

Pointcuts are used in the definition of advice. AspectJ has several different kinds of advice that define additional code that should run at join points. *Before advice* runs when a join point is reached and before the computation proceeds, i.e. it runs when computation reaches the method call and before the actual method starts running. *After advice* runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller. *Around advice* runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all. (Around advice and some variants of after advice are not discussed in this chapter.)

```
after(): move() {
 System.out.println( A figure element moved. );
}
```

### Exposing Context in Pointcuts

Pointcuts can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations. In the following code, the pointcut exposes three values from calls to `setXY`: the `FigureElement` receiving the call, the new value for `x` and the new value for `y`. The advice then prints the figure element that was moved and its new `x` and `y` coordinates after each `setXY` method call.

```
pointcut setXY(FigureElement fe, int x, int y):
   call(void FigureElement.setXY(int, int))
   && target(fe)
   && args(x, y);

after(FigureElement fe, int x, int y): setXY(fe, x, y) {
   System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

## Introduction

Introduction is AspectJ's form for modifying classes and their hierarchy. Introduction adds new members to classes and alters the inheritance relationship between classes. Unlike advice that operates primarily dynamically, introduction operates statically, at compilation time. Introduction changes the declaration of classes, and it is these changed classes that are inherited, extended or instantiated by the rest of the program.

Consider the problem of adding a new capability to some existing classes that are already part of a class heirarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to *each affected class* a method that implements this interface.

AspectJ can do better. The new capability is a crosscutting concern because it affects multiple classes. Using AspectJ's introduction form, we can introduced into existing classes, the methods or fields that are necessary to implement the new capability.

Suppose we want to have `Screen` objects observe changes to `Point` objects, where `Point` is an existing class. We can implement this by introducing into the class `Point` an instance field, `observers`, that keeps track of the `Screen` objects that are observing `Point`s. Observers are added or removed with the static methods `addObserver` and `removeObserver`. The pointcut `changes` defines what we want to observe, and the after advice defines what we want to do when we observe a change. Note that neither `Screen`'s nor `Point`'s code has to be modified, and that all the changes needed to support this new capability are local to this aspect.

```
aspect PointObserving {

   private Vector Point.observers = new Vector();

   public static void addObserver(Point p, Screen s) {
      p.observers.add(s);
   }
```

```
   public static void removeObserver(Point p, Screen s) {
      p.observers.remove(s);
   }

   pointcut changes(Point p): target(p) && call(void Point.set*(int));

   after(Point p): changes(p) {
      Iterator iter = p.observers.iterator();
      while ( iter.hasNext() ) {
         updateObserver(p, (Screen)iter.next());
      }
   }

   static void updateObserver(Point p, Screen s) {
      s.display(p);
   }
}
```

### Aspect Declarations

An *aspect* is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers. The crosscutting implementation is provided in terms of pointcuts, advice and introductions. Only aspects may include advice, so while AspectJ may define crosscutting effects, the declaration of those effects is localized.

The next three sections present the use of aspects in increasingly sophisticated ways. Development aspects are easily removed from production builds. Production aspects are intended to be used in both development and in production, but tend to affect only a few classes. Finally, reusable aspects require the most experience to get right.

## Development Aspects

This section presents examples of aspects that can be used during development of Java applications. These aspects facilitate debugging, testing and performance tuning work. The aspects define behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application. Using AspectJ makes it possible to cleanly modularize this kind of functionality, thereby making it possible to easily enable and disable the functionality when desired.

### Tracing, Logging, and Profiling

This first example shows how to increase the visibility of the internal workings of a program. It is a simple tracing aspect that prints a message at specified method calls. In our figure editor example, one such aspect might simply trace whenever points are drawn.

```
aspect SimpleTracing {
  pointcut tracedCall():
    call(void FigureElement.draw(GraphicsContext));

  before(): tracedCall() {
```

```
    System.out.println("Entering: " + thisJoinPoint);
  }
}
```

This code makes use of the `thisJoinPoint` special variable. Within all advice bodies this variable is bound to an object that describes the current join point. The effect of this code is to print a line like the following every time a figure element receives a `draw` method call:

```
Entering: call(void FigureElement.draw(GraphicsContext))
```

To understand the benefit of coding this with AspectJ consider changing the set of method calls that are traced. With AspectJ, this just requires editing the definition of the `tracedCalls` pointcut and recompiling. The individual methods that are traced do not need to be edited.

When debugging, programmers often invest considerable effort in figuring out a good set of trace points to use when looking for a particular kind of problem. When debugging is complete or appears to be complete it is frustrating to have to lose that investment by deleting trace statements from the code. The alternative of just commenting them out makes the code look bad, and can cause trace statements for one kind of debugging to get confused with trace statements for another kind of debugging.

With AspectJ it is easy to both preserve the work of designing a good set of trace points and disable the tracing when it isn t being used. This is done by writing an aspect specifically for that tracing mode, and removing that aspect from the compilation when it is not needed.

This ability to concisely implement and reuse debugging configurations that have proven useful in the past is a direct result of AspectJ modularizing a crosscutting design element the set of methods that are appropriate to trace when looking for a given kind of information.

### Profiling and Logging

Our second example shows you how to do some very specific profiling. Although many sophisticated profiling tools are available, and these can gather a variety of information and display the results in useful ways, you may sometimes want to profile or log some very specific behavior. In these cases, it is often possible to write a simple aspect similar to the ones above to do the job.

For example, the following aspect[1] will count the number of calls to the `rotate` method on a `Line` and the number of calls to the `set*` methods of a `Point` that happen within the control flow of those calls to `rotate`:

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }

  before(): call(void Point.set*(int)) &&
            cflow(call(void Line.rotate(double))) {
    setCount++;
  }
}
```

Aspects have an advantage over standard profiling or logging tools because they can be programmed to ask very specific and complex questions like, "How many times is the `rotate` method defined on `Line` objects called, and how many times are methods defined on `Point` objects whose name begins with `` `set' `` called in fulfilling those rotate calls"?

## Pre− and Post−Conditions

Many programmers use the "Design by Contract" style popularized by Bertand Meyer in *Object−Oriented Software Construction, 2/e*. In this style of programming, explicit pre−conditions test that callers of a method call it properly and explicit post−conditions test that methods properly do the work they are supposed to.

AspectJ makes it possible to implement pre− and post−condition testing in modular form. For example, this code

```
aspectPointBoundsChecking {

  pointcut setX(int x): (call(void FigureElement.setXY(int, int)) ||
                         call(void Point.setX(int)))
                        && args(x, ..);

  pointcut setY(int y): (call(void FigureElement.setXY(int, int)) ||
                         call(void Point.setY(int)))
                        && args(y, ..);

  before(int x): setX(x) {
    if ( x < MIN_X || x > MAX_X )
      throw new IllegalArgumentException("x is out of bounds.");
  }

  before(int y): setY(y) {
    if ( y < MIN_Y || y > MAX_Y )
      throw new IllegalArgumentException("y is out of bounds.");
  }

}
```

implements the bounds checking aspect of pre−condition testing for operations that move points. Notice that the `setX` pointcut refers to all the operations that can set a point's `x` coordinate; this includes the `setX` method, as well as half of the `setXY` method. In this sense the `setX` pointcut can be seen as involving very fine−grained crosscutting it names the the `setX` method and half of the `setXY` method.

Even though pre− and post−condition testing aspects can often be used only during testing, in some cases developers may wish to include them in the production build as well. Again, because AspectJ makes it possible to modularize these crosscutting concerns cleanly, it gives developers good control over this decision.

## Contract Enforcement

The property−based crosscutting mechanisms can be very useful in defining more sophisticated contract enforcement. One very powerful use of these mechanisms is to identify method calls that, in a correct program, should not exist. For example, the following aspect enforces the constraint that only the well−known

factory methods can add an element to the registry of figure elements. Enforcing this constraint ensures that no figure element is added to the registry more than once.

```
static aspect RegistrationProtection {

  pointcut register(): call(void Registry.register(FigureElement));

  pointcut canRegister(): withincode(static * FigureElement.make*(..));

  before(): register() && !canRegister() {
    throw new IllegalAccessException("Illegal call " + thisJoinPoint);
  }
}
```

This aspect uses the withincode primitive pointcut to denote all join points that occur within the body of the factory methods on `FigureElement` (the methods with names that begin with "`make`"). This is a property−based pointcut because it identifies join points based not on their signature, but rather on the property that they occur specifically within the code of another method. The before advice declaration effectively says signal an error for any calls to register that are not within the factory methods.

### Configuration Management

Configuration management for aspects can be handled using a variety of make−file like techniques. To work with optional aspects, the programmer can simply define their make files to either include the aspect in the call to the AspectJ compiler or not, as desired.

Developers who want to be certain that no aspects are included in the production build can do so by configuring their make files so that they use a traditional Java compiler for production builds. To make it easy to write such make files, the AspectJ compiler has a command−line interface that is consistent with ordinary Java compilers.

## Production Aspects

This section presents examples of aspects that are inherently intended to be included in the production builds of an application. Production aspects tend to add functionality to an application rather than merely adding more visibility of the internals of a program. Again, we begin with name−based aspects and follow with property−based aspects. Name−based production aspects tend to affect only a small number of methods. For this reason, they are a good next step for projects adopting AspectJ. But even though they tend to be small and simple, they can often have a significant effect in terms of making the program easier to understand and maintain.

### Change Monitoring

The first example production aspect shows how one might implement some simple functionality where it is problematic to try and do it explicitly. It supports the code that refreshes the display. The role of the aspect is

to maintain a dirty bit indicating whether or not an object has moved since the last time the display was refreshed.

Implementing this functionality as an aspect is straightforward. The `testAndClear` method is called by the display code to find out whether a figure element has moved recently. This method returns the current state of the dirty flag and resets it to false. The pointcut `move` captures all the method calls that can move a figure element. The after advice on `move` sets the dirty flag whenever an object moves.

```
aspect MoveTracking {

  private static boolean dirty = false;

  public static boolean testAndClear() {
    boolean result = dirty;
    dirty = false;
    return result;
  }

  pointcut move(): call(void FigureElement.setXY(int, int)) ||
                   call(void Line.setP1(Point))             ||
                   call(void Line.setP2(Point))             ||
                   call(void Point.setX(int))               ||
                   call(void Point.setY(int));

  after() returning: move() {
    dirty = true;
  }
}
```

Even this simple example serves to illustrate some of the important benefits of using AspectJ in production code. Consider implementing this functionality with ordinary Java: there would likely be a helper class that contained the `dirty` flag, the `testAndClear` method, as well as a `setFlag` method. Each of the methods that could move a figure element would include a call to the `setFlag` method. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case.

The AspectJ implementation has several advantages over the standard implementation:

*The structure of the crosscutting concern is captured explicitly.* The moves pointcut clearly states all the methods involved, so the programmer reading the code sees not just individual calls to `setFlag`, but instead sees the real structure of the code. As shown in the figure, Emacs Screenshot, the IDE support included with AspectJ will automatically remind the programmer that this aspect advises each of the methods involved.[2] The text in `[Square Brackets]` following the method declarations is automatically generated, and serves to remind the programmer of the aspects that crosscut the method. The editor also provide commands to jump to the advice from the method and vice−versa.

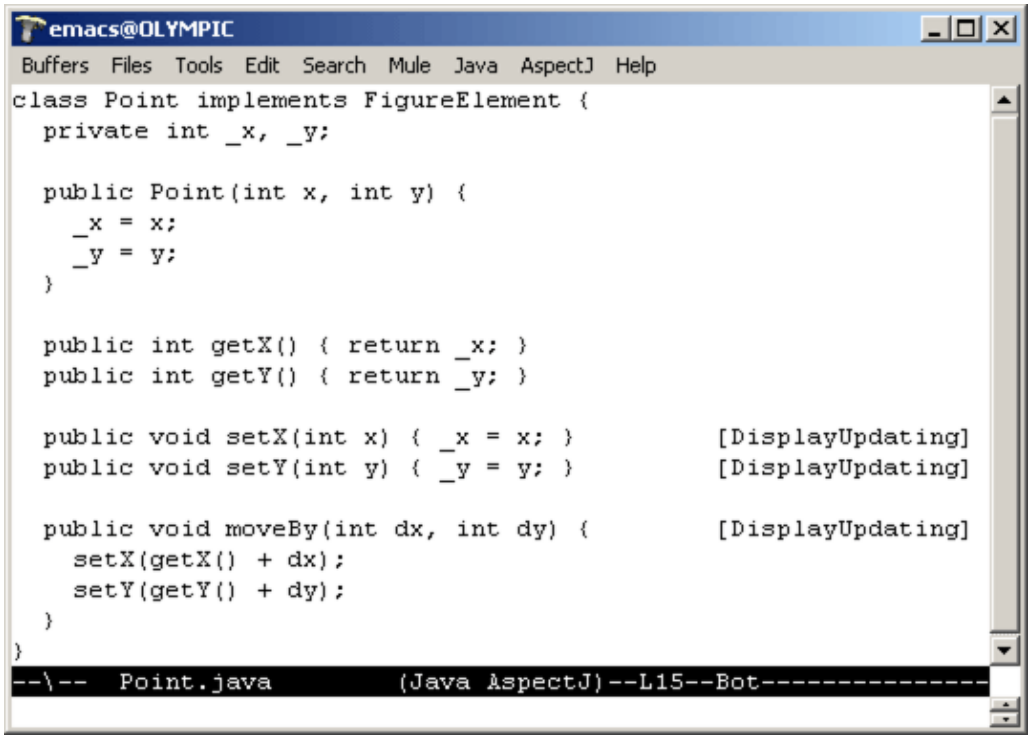
*Evolution is easier.* If, for example, the aspect needs to be revised to record not just that some figure element moved, but rather to record exactly which figure elements moved, the change would be entirely local to the aspect. The pointcut would be updated to expose the object being moved, and the advice would be updated to record that object. The paper *An Overview of AspectJ*, presented at ECOOP 2001, presents a detailed discussion of various ways this aspect could be expected evolve.)

*The functionality is easy to plug in and out.* Just as with development aspects, production aspects may need to be removed from the system, either because the functionality is no longer needed at all, or because it is not

needed in certain configurations of a system. Because the functionality is modularized in a single aspect this is easy to do.

*The implementation is more stable.* If, for example, the programmer adds a subclass of `Line` that overrides the existing methods, this advice in this aspect will still apply. In the ordinary Java implementation the programmer would have to remember to add the call to `setFlag` in the new overriding method. This benefit is often even more compelling for property−based aspects (see the section Providing Consistent Behavior).

**Figure 1.1. Using the AspectJ−aware extension to Emacs.**




## Context Passing

The crosscutting structure of context passing can be a significant source of complexity in Java programs. Consider implementing functionality that would allow a client of the figure editor (a program client rather than a human) to set the color of any figure elements that are created. Typically this requires passing a color, or a color factory, from the client, down through the calls that lead to the figure element factory. All programmers are familiar with the inconvenience of adding a first argument to a number of methods just to pass this kind of context information.

Using AspectJ, this kind of context passing can be implemented in a modular way. The following code adds after advice that runs only when the factory methods of `Figure` are called in the control flow of a method on a `ColorControllingClient`.

```
aspect ColorControl {

  pointcut CCClientCflow(ColorControllingClient client):
    cflow(call(* * (..)) && target(client));

  pointcut make(): call(FigureElement Figure.make*(..));
```

```
  after (ColorControllingClient c) returning (FigureElement fe):
        make() && CCClientCflow(c) {

    fe.setColor(c.colorFor(fe));
  }
}
```

This aspect affects only a small number of methods, but note that the non−AOP implementation of this functionality might require editing many more methods, specifically, all the methods in the control flow from the client to the factory. This is a benefit common to many property−based aspects while the aspect is short and affects only a modest number of benefits, the complexity the aspect saves is potentially much larger.

## Providing Consistent Behavior

This example shows how a property−based aspect can be used to provide consistent handling of functionality across a large set of operations. This aspect ensures that all public methods of the `com.xerox` package log any errors they throw to their caller. The `publicMethodCall` pointcut captures the public method calls of the package, and the after advice runs whenever one of those calls returns throwing an exception. The advice logs the exception and then the throw resumes.

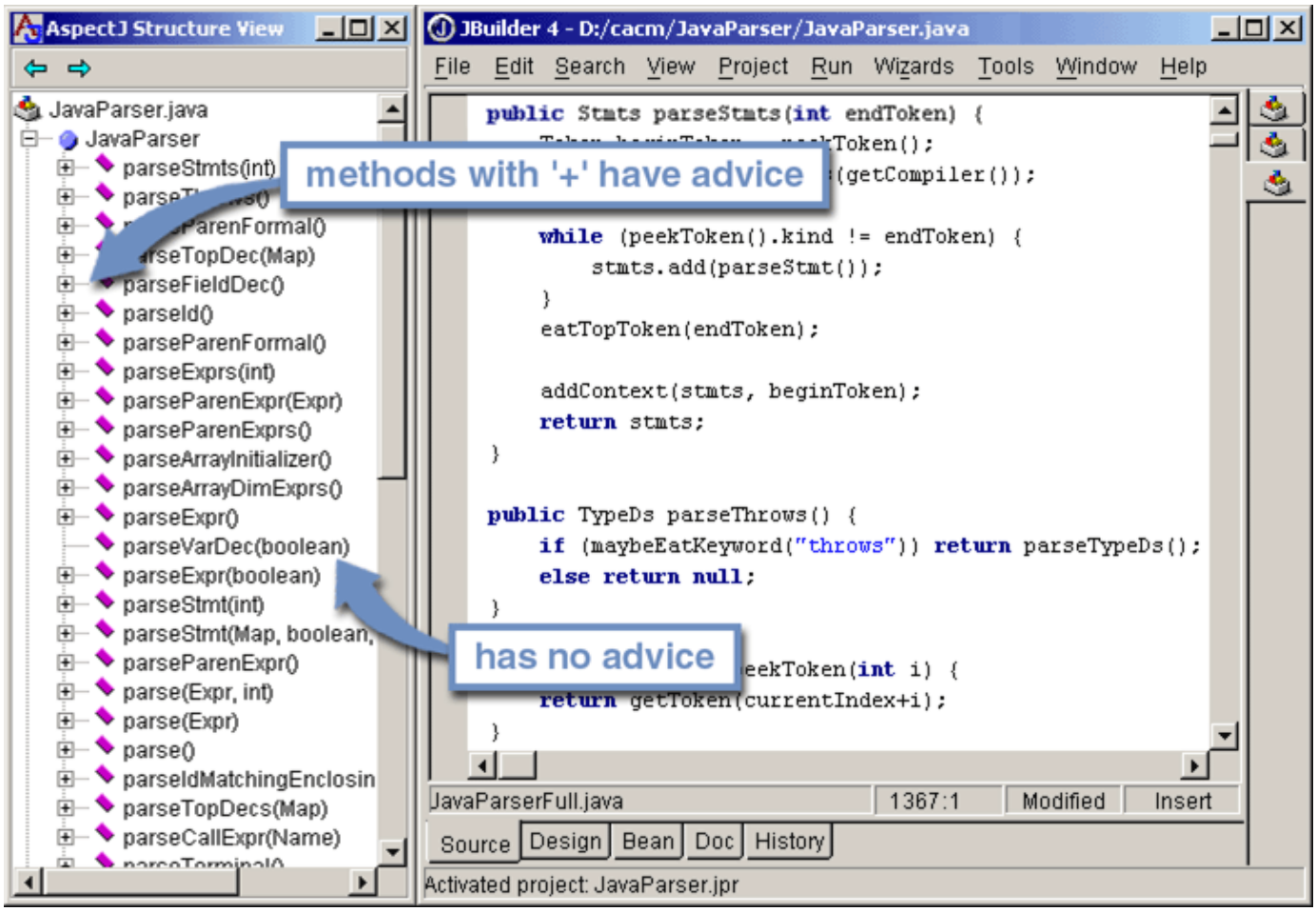
```
aspect PublicErrorLogging {
   Log log = new Log();

   pointcut publicMethodCall(): call(public * com.xerox.*.*(..));

   after() throwing (Error e): publicMethodCall() { log.write(e); }
}
```

In some cases this aspect can log an exception twice. This happens if code inside the `com.xerox` package itself calls a public method of the package. In that case this code will log the error at both the outermost call into the `com.xerox` package and the re−entrant call. The `cflow` primitive pointcut can be used in a nice way to exclude these re−entrant calls:

```
after() throwing (Error e): publicMethodCall() &&
                            !cflow(publicMethodCall()) {
   log.write(e);
}
```

The following aspect is taken from work on the AspectJ compiler. The aspect advises about 35 methods in the `JavaParser` class. The individual methods handle each of the different kinds of elements that must be parsed. They have names like `parseMethodDec`, `parseThrows`, and `parseExpr`.

**Figure 1.2. Examining the partially re−factored code with the AJDE extension to JBuilder.**




```
aspect ContextFilling {
   pointcut parse(JavaParser jp):
         call(* JavaParser.parse*(..))
         && target(jp)
         && !call(Stmt parseVarDec(boolean)); // var decs
                                              // are tricky

   around(JavaParser jp) returns ASTObject: parse(jp) {
      Token beginToken = jp.peekToken();
      ASTObject ret = proceed(jp);
      if (ret != null) jp.addContext(ret, beginToken);
      return ret;
   }
}
```

This example exhibits a property found in many aspects with large property−based pointcuts. In addition to a general property based pattern `calls(* jp.parse*(..))` it includes an exception to the pattern `!calls(Stmt parseVarDec(boolean))`. The exclusion of `parseVarDec` happens because the parsing of variable declarations in Java is too complex to fit with the clean pattern of the other `parse*` methods. Even with the explicit exclusion this aspect is a clear expression of a clean crosscutting modularity. Namely that all `parse*` methods that return `ASTObjects`, except for `parseVarDec` share a common behavior for establishing the parse context of their result.

The process of writing an aspect with a large property−based pointcut, and of developing the appropriate exceptions can clarify the structure of the system. This is especially true, as in this case, when refactoring

existing code to use aspects. When we first looked at the code for this aspect, we were able to use the IDE support provided in AJDE for JBuilder to see what methods the aspect was advising compared to our manual coding. We used the AJDE structure view shown in the figure The AJDE Extension to JBuilder and scrolled through the code. Method names marked with + are advised; those without are not. We quickly discovered that there were a dozen places where the aspect advice was in effect but we had not manually inserted the required functionality. Two of these were bugs in our prior non−AOP implementation of the parser. The other ten were needless performance optimizations. So, here, refactoring the code to express the crosscutting structure of the aspect explicitly made the code more concise and eliminated latent bugs.

# Static Crosscutting: Introduction

Up until now we have only seen constructs that allow us to implement dynamic crosscutting, crosscutting that changes the way a program executes. AspectJ also allows us to implement static crosscutting, crosscutting that affects the static structure of our progams. This is done using forms called introduction.

An *introduction* is a member of an aspect, but it defines or modifies a member of another type (class). With introduction we can

- add methods to an existing class
- add fields to an existing class
- extend an existing class with another
- implement an interface in an existing class
- convert checked exceptions into unchecked exceptions

Suppose we want to change the class `Point` to support cloning. By using introduction, we can add that capability. The class itself doesn't change, but its users (here the method `main`) may. In the example below, the aspect `CloneablePoint` does three things:

1. declares that the class `Point` implements the interface `Cloneable`,
2. declares that the methods in `Point` whose signature matches `Object clone()` should have their checked exceptions converted into unchecked exceptions, and
3. adds a method that overrides the method `clone` in `Point`, which was inherited from `Object`.

```
class Point {
   private int x, y;
```

```
   Point(int x, int y) { this.x = x; this.y = y; }

   int getX() { return this.x; }
   int getY() { return this.y; }

   void setX(int x) { this.x = x; }
   void setY(int y) { this.y = y; }

   public static void main(String[] args) {

      Point p = new Point(3,4);
      Point q = (Point) p.clone();
   }
}

aspect CloneablePoint {
   declare parents: Point implements Cloneable;

   declare soft: CloneNotSupportedException: execution(Object clone());

   Object Point.clone() { return super.clone(); }
}
```

Introduction is a powerful mechanism for capturing crosscutting concerns because it not only changes the behavior of components in an application, but also changes their relationship.

# Conclusion

AspectJ is a simple and practical aspect−oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns.

Adoption of AspectJ into an existing Java development project can be a straightforward task and incremental task. One path is to begin by using only development aspects, going on to using production aspects and then reusable aspects after building up experience with AspectJ. Adoption can follow other paths as well. For example, some developers will benefit from using production aspects right away. Others may be able to write clean reusable aspects almost right away.

Aspect enables both name−based and property based crosscutting. Aspects that use name−based crosscutting tend to affect a small number of other classes. But despite their small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property−based crosscutting can have small or large scale.

Using AspectJ results in clean well−modularized implementations of crosscutting concerns. When written as an AspectJ aspect the structure of a crosscutting concern is explicit and easy to understand. Aspects are also highly modular, making it possible to develop plug−and−play implementations of crosscutting functionality.

AspectJ provides more functionality than was covered by this short introduction. The next chapter, The AspectJ Language, covers in detail all the features of the AspectJ language. The following chapter, Examples, then presents some carefully chosen examples that show you how AspectJ might be used. We recommend that you read the next two chapters carefully before deciding to adopt AspectJ into a project.

[1] Since aspects are by default singleton aspects, i.e. there is only one instance of the aspect, fields in a singleton aspect are similar to static fields in class.

[2] Support for the JBuilder and Forte for Java IDEs, as well as for GNU Emacs/XEmacs, are included in the AspectJ distribution.

# Chapter 2. The AspectJ Language

**Table of Contents**



# Introduction

The previous chapter, Getting Started with AspectJ, was a brief overview of the AspectJ language. You should read this chapter to understand AspectJ's syntax and semantics. It covers the same material as the previous chapter, but more completely and in much more detail.

We will start out by looking at an example aspect that we'll build out of a pointcut, an introduction, and two pieces of advice. This example aspect will gives us something concrete to talk about.

# The Anatomy of an Aspect

This lesson explains the parts of AspectJ's aspects. By reading this lesson you will have an overview of what's in an aspect and you will be exposed to the new terminology introduced by AspectJ.

## An Example Aspect

Here's an example of an aspect definition in AspectJ:

```
 1 aspect FaultHandler {
```

```
 2
 3   private boolean Server.disabled = false;
 4
 5   private void reportFault() {
 6     System.out.println("Failure! Please fix it.");
 7   }
 8
 9   public static void fixServer(Server s) {
10     s.disabled = false;
11   }
12
13   pointcut services(Server s): target(s) && call(public * *(..));
14
15   before(Server s): services(s) {
16     if (s.disabled) throw new DisabledException();
17   }
18
19   after(Server s) throwing (FaultException e): services(s) {
20     s.disabled = true;
21     reportFault();
22   }
23 }
```

The `FaultHandler` consists of one variable introduced onto `Server` (line 03), two methods (lines 05−07 and 09−11), one pointcut (line 13), and two pieces of advice (lines 15−17 and 19−22).

This covers the basics of what aspects can contain. In general, aspects consist of an association with other program entities, ordinary variables and methods, pointcuts, introductions, and advice, where advice may be before, after or around advice. The remainder of this lesson focuses on those crosscut−related constructs.

## Pointcuts

AspectJ's pointcuts define collections of events, i.e. interesting points in the execution of a program. These events, or points in the execution, can be method or constructor invocations and executions, handling of exceptions, field assignments and accesses, etc. Take, for example, the pointcut declaration in line 13:

```
pointcut services(Server s): target(s) && call(public * *(..))
```

This pointcut, named `services`, picks out those points in the execution of the program when instances of the `Server` class have their public methods called.

The idea behind this pointcut in the `FaultHandler` aspect is that fault−handling−related behavior must be triggered on the calls to public methods. For example, the server may be unable to proceed with the request because of some fault. The calls of those methods are, therefore, interesting events for this aspect, in the sense that certain fault−related things will happen when these events occur.

Part of the context in which the events occur is exposed by the formal parameters of the pointcut. In this case, that consists of objects of type server. That formal parameter is then being used on the right hand side of the declaration in order to identify which events the pointcut refers to. In this case, a pointcut picking out join points where a Server is the target of some operation (target(s)) is being composed (&&, meaning and) with a pointcut picking out call join points (call(...)). The calls are identified by signatures that can include wild cards. In this case, there are wild cards in the return type position (first *), in the name position (second *) and in the argument list position (..); the only concrete information is given by the qualifier public.

### What else?

Pointcuts define arbitrarily large sets of points in the execution of a program. But they use only a finite number of *kinds* of points. Those kinds of points correspond to some of the most important concepts in Java. Here is an incomplete list: method invocation, method execution, exception handling, instantiation, constructor execution. Each of these has a specific syntax that you will learn about in other parts of this guide.

## Advice

Advice defines pieces of aspect implementation that execute at join points picked out by a pointcut. For example, the advice in lines 15−17 specifies that the following piece of code

```
{
  if (s.disabled) throw new DisabledException();
}
```

is executed when instances of the Server class have their public methods called, as specified by the pointcut services. More specifically, it runs when those calls are made, just before the corresponding methods are executed.

The advice in lines 19−22 defines another piece of implementation that is executed on the same pointcut:

```
{
  s.disabled = true;
  reportFault();
}
```

But this second method executes whenever those operations throw exception of type `FaultException`.

### What else?

There are two other variations of after advice: upon successful return and upon return, either successful or with an exception. There is also a third kind of advice called around. You will see those in other parts of this guide.

# Join Points and Pointcuts

Consider the following Java class:

```
class Point {
    private int x, y;

    Point(int x, int y) { this.x = x; this.y = y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }

    int getX() { return x; }
    int getY() { return y; }
}
```

In order to get an intuitive understanding of AspectJ's pointcuts, let's go back to some of the basic principles of Java. Consider the following a method declaration in class Point:

```
void setX(int x) { this.x = x; }
```

What this piece of program states is that when an object of type Point has a method called setX with an integer as the argument called on it, then the method body { this.x = x; } is executed. Similarly, the constructor given in that class states that when an object of type Point is instantiated through a constructor with two integers as arguments, then the constructor body { this.x = x; this.y = y; } is executed.

One pattern that emerges from these descriptions is when something happens, then something gets executed. In object−oriented programs, there are several kinds of "things that happen" that are determined by the language. We call these the join points of Java. Join points comprised method calls, method executions, instantiations, constructor executions, field references and handler executions. (See the quick reference for complete listing.)

Pointcuts pick out these join points. For example, the pointcut

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

describes the calls to `setX(int)` or `setY(int)` methods of any instance of Point. Here's another example:

```
pointcut ioHandler(): within(MyClass) && handler(IOException);
```

This pointcut picks out the join points at which exceptions of type IOException are handled inside the code defined by class MyClass.

Pointcuts consist of a left−hand side and a right−hand side, separated by a colon. The left−hand side defines the pointcut name and the pointcut parameters (i.e. the data available when the events happen). The right−hand side defines the events in the pointcut.

Pointcuts can then be used to define aspect code in advice, as we will see later. But first let's see what types of events can be captured and how they are described in AspectJ.

## Designators

Here are examples of designators of

*when a particular method body executes*

```
execution(void Point.setX(int))
```

*when a method is called*

```
call(void Point.setX(int))
```

*when an exception handler executes*

```
handler(ArrayOutOfBoundsException)
```

*when the object currently executing (i.e. `this`) is of type `SomeType`*

```
this(SomeType)
```

*when the target object is of type `SomeType`*

```
target(SomeType)
```

*when the executing code belongs to class `MyClass`*

```
within(MyClass)
```

*when the join point is in the control flow of a call to a `Test`'s no−argument `main` method*

```
cflow(void Test.main())
```

Designators compose through the operations `or` ("`||`"), `and` ("&&") and `not` ("`!`").

- It is possible to use wildcards. So

  1. `execution(* *(..))`
  2. `call(* set(..))`

  means (1) all the executions of methods with any return and parameter types and (2) method calls of set methods with any return and parameter types −− in case of overloading there may be more than one; this designator picks out all of them.

- You can select elements based on types. For example,

  1. `execution(int *())`
  2. `call(* setY(long))`
  3. `call(* Point.setY(int))`
  4. `call(*.new(int, int))`

  means (1) all executions of methods with no parameters, returning an `int` (2) the calls of `setY` methods that take a `long` as an argument, regardless of their return type or defining type, (3) the calls of class `Point`'s `setY` methods that take an `int` as an argument, regardless of the return type, and (4) the calls of all classes' constructors that take two `int`s as arguments.

- You can compose designators. For example,

  1.

```
target(Point) && call(int *())
```

2. ```
   call(* *(..)) && (within(Line) || within(Point))
   ```

3. ```
   within(*) && execution(*.new(int))
   ```

4. ```
   this(*) && !this(Point) && call(int *(..))
   ```

means (1) all calls to methods received by instances of class `Point`, with no parameters, returning an `int`, (2) calls to any method where the call is made from the code in `Point`'s or `Line`'s type declaration, (3) executions of constructors of all classes, that take an `int` as an argument, and (4) all method calls of any method returning an `int`, from all objects except `Point` objects to any other objects.

- You can select methods and constructors based on their modifiers and on negations of modifiers. For example, you can say:

  1. ```
     call(public * *(..))
     ```

  2. ```
     execution(!static * *(..))
     ```

  3. ```
     execution(public !static * *(..))
     ```

  which means (1) all invocation of public methods, (2) all executions of non−static methods, and (3) all signatures of the public, non−static methods.

- Designators can also deal with interfaces. For example, given the interface

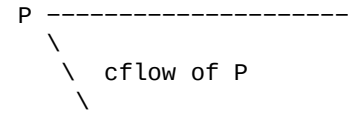
  ```
  interface MyInterface { ... }
  ```

  the designator `call(* MyInterface.*(..))` picks out the call join points for methods defined by the interface `MyInterface` (or its superinterfaces).

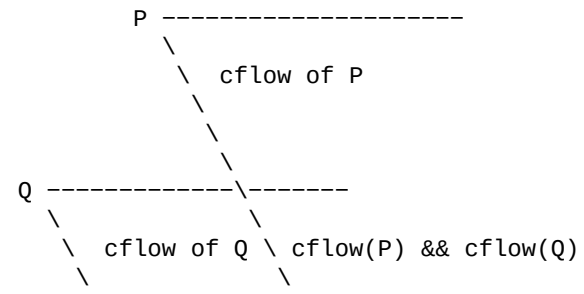
## Pointcut composition

Pointcuts are put together with the operators and (spelled &&), or (spelled `||`), and not (spelled `!`). This allows the creation of very powerful pointcuts from the simple building blocks of primitive pointcuts. This composition can be somewhat confusing when used with primitive pointcuts like cflow and cflowbelow. Here's an example:

`cflow(`*P*`)` picks out the join points in the control flow of the join points picked out by *P*. So, pictorially:

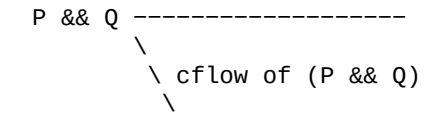



What does `cflow(P) && cflow(Q)` pick out? Well, it picks out those join points that are in both the control flow of *P* and in the control flow of *Q*. So...




Note that *P* and *Q* might not have any join points in common... but their control flows might have join points in common.

But what does `cflow(P && Q)` mean? Well, it means the control flow of those join points that are both picked out by *P* picked out by *Q*.




and if there are *no* join points that are both picked by *P* and picked out by *Q*, then there's no chance that there are any join points in the control flow of `(P && Q)`.

Here's some code that expresses this.

```
public class Test {
    public static void main(String[] args) {
        foo();
    }
    static void foo() {
        goo();
    }
    static void goo() {
        System.out.println("hi");
    }
}

aspect A  {

    pointcut fooPC(): execution(void Test.foo());
    pointcut gooPC(): execution(void Test.goo());
    pointcut printPC(): call(void java.io.PrintStream.println(String));

    before(): cflow(fooPC()) && cflow(gooPC()) && printPC() {
        System.out.println("should occur");
    }

    static before(): cflow(fooPC() && gooPC()) && printPC() {
```

```
        System.out.println("should not occur");
    }

}
```

## Pointcut Parameters

Consider, for example, the first pointcut you've seen here,

```
  pointcut setter(): target(Point) &&
                     (call(void setX(int)) ||
                      call(void setY(int)));
```

As we've seen before, the right−hand side of the pointcut picks out the calls to `setX(int)` or `setY(int)` methods where the target is any object of type `Point`. On the left−hand side, the pointcut is given the name "setters" and no parameters. An empty parameter list means that when those events happen no context is immediately available. But consider this other version of the same pointcut:

```
  pointcut setter(Point p): target(p) &&
                            (call(void setX(int)) ||
                             call(void setY(int)));
```

This version picks out exactly the same calls. But in this version, the pointcut has one parameter of type `Point`. This means that when the events described on the right−hand side happen, a `Point` object, named by a parameter named "p", is available. According to the right−hand side of the pointcut, that `Point` object in the pointcut parameters is the object that receives the calls.

Here's another example that illustrates the flexible mechanism for defining pointcut parameters:

```
  pointcut testEquality(Point p): target(Point) &&
                                  args(p) &&
                                  call(boolean equals(Object));
```

This pointcut also has a parameter of type `Point`. Similarly to the "setters" pointcut, this means that when the events described on the right−hand side happen, a `Point` object, named by a parameter named "p", is available. But in this case, looking at the right−hand side, we find that the object named in the parameters is not the target `Point` object that receives the call; it's the argument (of type Point) passed to the "equals" method on some other target Point object. If we wanted access to both objects, then the pointcut definition that would define target `Point p1` and argument `Point p2` would be

```
  pointcut testEquality(Point p1, Point p2): target(p1) &&
                                             args(p2) &&
                                             call(boolean equals(Object));
```

Let's look at another variation of the "setters" pointcut:

```
pointcut setter(Point p, int newval): target(p) &&
                                      args(newval) &&
                                      (call(void setX(int)) ||
                                       call(void setY(int)));
```

In this case, a `Point` object and an integer value are available when the calls happen. Looking at the events definition on the right−hand side, we find that the `Point` object is the object receiving the call, and the

integer value is the argument of the method .

The definition of pointcut parameters is relatively flexible. The most important rule is that when each of those events defined in the right−hand side happen, all the pointcut parameters must be bound to some value. So, for example, the following pointcut definition will result in a compilation error:

```
pointcut xcut(Point p1, Point p2):
    (target(p1) && call(void setX(int))) ||
    (target(p2) && call(void setY(int)));
```

The right−hand side establishes that this pointcut picks out the call join points consisting of the `setX(int)` method called on a point object, or the `setY(int)` method called on a point object. This is fine. The problem is that the parameters definition tries to get access to two point objects. But when `setX(int)` is called on a point object, there is no other point object to grab! So in that case, the parameter `p2` is unbound, and hence, the compilation error.

## Example: `HandleLiveness`

The example below consists of two object classes (plus an exception class) and one aspect. Handle objects delegate their public, non−static operations to their `Partner` objects. The aspect `HandleLiveness` ensures that, before the delegations, the partner exists and is alive, or else it throws an exception.

```
class Handle {
  Partner partner = new Partner();

  public void foo() { partner.foo(); }
  public void bar(int x) { partner.bar(x); }

  public static void main(String[] args) {
    Handle h1 = new Handle();
    h1.foo();
    h1.bar(2);
  }
}

class Partner {
  boolean isAlive() { return true; }
  void foo() { System.out.println("foo"); }
  void bar(int x) { System.out.println("bar " + x); }
}

aspect HandleLiveness {
  before(Handle handle): target(handle) && call(public * *(..)) {
    if ( handle.partner == null  || !handle.partner.isAlive() ) {
      throw new DeadPartnerException();
    }
  }
}

class DeadPartnerException extends RuntimeException {}
```

# Advice

Advice defines pieces of aspect implementation that execute at well−defined points in the execution of the program. Those points can be given either by named pointcuts (like the ones you've seen above) or by anonymous pointcuts. Here is an example of an advice on a named pointcut:

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                       (call(void setX(int) ||
                                        call(void setY(int)));

before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 +
                       " to the new value " + newval);
}
```

And here is exactly the same example, but using an anonymous pointcut:

```
before(Point p1, int newval): target(p1) && args(newval)
                              (call(void setX(int)) ||
                               call(void setY(int))) {
    System.out.println("About to set something in " + p1 +
                       " to the new value " + newval);
}
```

Here are examples of the different advice:

```
before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
    if (!p.assertX(x)) return;
}
```

This before advice runs just before the execution of the actions associated with the events in the (anonymous) pointcut.

```
after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
    if (!p.assertX(x)) throw new PostConditionViolation();
}
```

This after advice runs just after each join point picked out by the (anonymous) pointcut, regardless of whether it returns normally or throws an exception.

```
after(Point p) returning(int x): target(p) && call(int getX()) {
    System.out.println("Returning int value " + x + " for p = " + p);
}
```

This after returning advice runs just after each join point picked out by the (anonymous) pointcut, but only if it returns normally. The return value can be accessed, and is named `x` here. After the advice runs, the return value is returned.

```
after() throwing(Exception e): target(Point) && call(void setX(int)) {
    System.out.println(e);
}
```

This after throwing advice runs just after each join point picked out by the (anonymous) pointcut, but only when it throws an exception of type `Exception`. Here the exception value can be accessed with the name `e`. The advice re−raises the exception after it's done.

```
void around(Point p, int x): target(p)
                          && args(x)
                          && call(void setX(int)) {
    if (p.assertX(x)) proceed();
    p.releaseResources();
}
```

This around advice traps the execution of the join point; it runs *instead* of the join point. The original action associated with the join point can be invoked through the special `proceed` call.

# Introduction

Introduction declarations add whole new elements in the given types, and so change the type hierarchy. Here are examples of introduction declarations:

```
  private boolean Server.disabled = false;
```

This privately introduces a field named `disabled` in `Server` and initializes it to `false`. Because it is declared `private`, only code defined in the aspect can access the field.

```
  public int Point.getX() { return x; }
```

This publicly introduces a method named `getX` in `Point`; the method returns an `int`, it has no arguments, and its body is return `x`. Because it is defined publically, any code can call it.

```
  public Point.new(int x, int y) { this.x = x; this.y = y; }
```

This publicly introduces a constructor in Point; the constructor has two arguments of type int, and its body is this.x = x; this.y = y;

```
  public int Point.x = 0;
```

This publicly introduces a field named x of type int in Point; the field is initialized to 0.

```
  declare parents: Point implements Comparable;
```

This declares that the `Point` class now implements the `Comparable` interface. Of course, this will be an error unless `Point` defines the methods of `Comparable`.

```
  declare parents: Point extends GeometricObject;
```

This declares that the `Point` class now extends the `GeometricObject` class.

An aspect can introduce several elements in at the same time. For example, the following declaration

```
  public String Point.name;
  public void Point.setName(String name) { this.name = name; }
```

publicly introduces both a field and a method into class `Point`. Note that the identifier "name" in the body of the method is bound to the "name" field in `Point`, even if the aspect defined another field called "name".

One declaration can introduce several elements in several classes as well. For example,

```
public String (Point || Line || Square).getName()  { return name; }
```

publicly introduces three methods, one in `Point`, another in Line and another in `Square`. The three methods have the same name (getName), no parameters, return a String, and have the same body (return name;). The purpose of introducing several elements in one single declaration is that their bodies are the same. The introduction is an error if any of `Point`, `Line`, or `Square` do not have a "name" field.

An aspect can introduce fields and methods (even with bodies) onto interfaces as well as classes.

## Introduction Scope

AspectJ allows private and package−protected (default) introduction in addition to public introduction. Private introduction means private in relation to the aspect, not necessarily the target type. So, if an aspect makes a private introduction of a field on a type

```
private int Foo.x;
```

Then code in the aspect can refer to Foo's x field, but nobody else can. Similarly, if an aspect makes a package−protected introduction,

```
int Foo.x;
```

then everything in the aspect's package (which may not be Foo's package) can access x.

## Example: `PointAssertions`

The example below consists of one class and one aspect. The aspect introduces all implementation that is related with assertions of the class. It privately introduces two methods in the class Point, namely assertX and assertY. It also advises the two set methods of Point with before declarations that assert the validity of the given values. The introductions are made privately because other parts of the program have no business accessing the assert methods. Only the code inside of the aspect can call those methods.

```
class Point  {
    int x, y;

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    public static void main(String[] args) {
        Point p = new Point();
        p.setX(3); p.setY(333);
    }
}

aspect PointAssertions {

    private boolean Point.assertX(int x) {
        return (x <= 100 && x >= 0);
    }
    private boolean Point.assertY(int y) {
        return (y <= 100 && y >= 0);
    }
```

```
    before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
        if (!p.assertX(x)) {
            System.out.println("Illegal value for x"); return;
        }
    }
    before(Point p, int y): target(p) && args(y) && call(void setY(int)) {
        if (!p.assertY(y)) {
            System.out.println("Illegal value for y"); return;
        }
    }
}
```

# Reflection

AspectJ provides a special reference variable, thisJoinPoint, that contains reflective information about the current join point for the advice to use. The thisJoinPoint variable can only be used in the context of advice, just like this can only be used in the context of non−static methods and variable initializers. In advice, thisJoinPoint is an object of type JoinPoint.

One way to use it is simply to print it out. Like all Java objects, thisJoinPoint has a toString() method that makes quick−and−dirty tracing easy.

```
class TraceNonStaticMethods {
    before(Point p): target(p) && call(* *(..)) {
        System.out.println("Entering " + thisJoinPoint + " in " + p);
    }
}
```

The type of thisJoinPoint includes a rich reflective class hierarchy of signatures, and can be used to access both static and dynamic information about join points. If, however, only the static information about the join point (such as the Signature) is desired, a lightweight join−point object is available from the thisJoinPointStaticPart special variable. This object is the same object you would get from

```
thisJoinPoint.getStaticPart()
```

The static part of a join point does not include dynamic information, such as the arguments, which can be accessed with

```
thisJoinPoint.getArgs()
```

But it has the performance benefit that repeated execution of the code containing `thisJoinPointStaticPart` (through, for example, separate method calls) will not result in repeated construction of the reflective object.

It is always the case that

```
thisJoinPointStaticPart == thisJoinPoint.getStaticPart()

thisJoinPoint.getKind() == thisJoinPointStaticPart.getKind()
thisJoinPoint.getSignature() == thisJoinPointStaticPart.getSignature()
thisJoinPoint.getSourceLocation() == thisJoinPointStaticPart.getSourceLocation()
```

# Chapter 3. Examples

**Table of Contents**



## About this Chapter

This chapter consists entirely of examples of AspectJ use.

The examples can be grouped into four categories:

| | |
|---|---|
| **technique** | Examples which illustrate how to use one or more features of the language. |
| **development** | Examples of using AspectJ during the development phase of a project. |
| **production** | Examples of using AspectJ to provide functionality in an application. |
| **reusable** | Examples of reuse of aspects and pointcuts. |

## Obtaining, Compiling and Running the Examples

The examples source code is part of AspectJ's documentation distribution which may be downloaded from the AspectJ download page.

Compiling most examples should be straightforward. Go the *`InstallDir`*`/examples` directory, and look for a `.lst` file in one of the example subdirectories. Use the `-arglist` option to `ajc` to compile the example. For instance, to compile the telecom example with billing, type

```
ajc -argfile telecom/billing.lst
```

To run the examples, your classpath must include the AspectJ run-time Java archive (`aspectjrt.jar`). You may either set the `CLASSPATH` environment variable or use the `-classpath` command line option to the Java interpreter:

```
(In Unix use a : in the CLASSPATH)
java -classpath ".:InstallDir/lib/aspectjrt.jar" telecom.billingSimulation

(In Windows use a ; in the CLASSPATH)
java -classpath ".;InstallDir/lib/aspectjrt.jar" telecom.billingSimulation
```

# Basic Techniques

This section presents two basic techniques of using AspectJ, one each from the two fundamental ways of capturing crosscutting concerns: with dynamic join points and advice, and with static introduction. Advice changes an application's behavior. Introduction changes both an application's behavior and its structure.

The first example, Join Points and `thisJoinPoint`, is about gathering and using information about the join point that has triggered some advice. The second example, Roles and Views Using Introduction, concerns changing an existing class hierarchy.

## Join Points and `thisJoinPoint`

(The code for this example is in *`InstallDir`*`/examples/tjp`.)

A join point is some point in the execution of a program together with a view into the execution context when that point occurs. Join points are picked out by pointcuts. When a join point is reached, before, after or around advice on that join point may be run.

When dealing with pointcuts that pick out join points of specific method calls, field gets, or the like, the advice will know exactly what kind of join point it is executing under. It might even have access to context given by its pointcut. Here, for example, since the only join points reached will be calls of a certain method, we can get the target and one of the args of the method directly.

```
before(Point p, int x): target(p)
                     && args(x)
                     && call(void setX(int)) {
    if (!p.assertX(x)) {
        System.out.println("Illegal value for x"); return;
    }
}
```

But sometimes the join point is not so clear. For instance, suppose a complex application is being debugged, and one would like to know when any method in some class is being executed. Then, the pointcut

```
pointcut execsInProblemClass(): within(ProblemClass)
                             && execution(* *(..));
```

will select all join points where a method defined within the class `ProblemClass` is being executed. But advice executes when a particular join point is matched, and so the question, "Which join point was matched?" naturally arises.

Information about the join point that was matched is available to advice through the special variable `thisJoinPoint`, of type `org.aspectj.lang.JoinPoint`. This class provides methods that return

- the kind of join point that was matched
- the source location of the current join point
- normal, short and long string representations of the current join point
- the actual argument(s) to the method or field selected by the current join point
- the signature of the method or field selected by the current join point
- the target object
- the currently executing object

- a reference to the static portion of the object representing the current join point. This is also available through the special variable `thisJoinPointStaticPart`.

### The `Demo` class

The class `tjp.Demo` in `tjp/Demo.java` defines two methods `foo` and `bar` with different parameter lists and return types. Both are called, with suitable arguments, by `Demo`'s `go` method which was invoked from within its `main` method.

```
public class Demo {

    static Demo d;

    public static void main(String[] args){
        new Demo().go();
    }

    void go(){
        d = new Demo();
        d.foo(1,d);
        System.out.println(d.bar(new Integer(3)));
    }

    void foo(int i, Object o){
        System.out.println("Demo.foo(" + i + ", " + o + ")\n");
    }


    String bar (Integer j){
        System.out.println("Demo.bar(" + j + ")\n");
        return "Demo.bar(" + j  + ")";
    }

}
```

### The Aspect `GetInfo`

This aspect uses around advice to intercept the execution of methods `foo` and `bar` in `Demo`, and prints out information garnered from `thisJoinPoint` to the console.

#### Defining the scope of a pointcut

The pointcut `goCut` is defined as `cflow(this(Demo)) && execution(void go())` so that only executions made in the control flow of `Demo.go` are intercepted. The control flow from the method `go` includes the execution of `go` itself, so the definition of the around advice includes `!execution(* go())` to exclude it from the set of executions advised.

#### Printing the class and method name

The name of the method and that method's defining class are available as parts of the Signature, found using the method `getSignature` of either `thisJoinPoint` or `thisJoinPointStaticPart`.

```
aspect GetInfo {

   static final void println(String s){ System.out.println(s); }

   pointcut goCut(): cflow(this(Demo) && execution(void go()));

   pointcut demoExecs(): within(Demo) && execution(* *(..));

   Object around(): demoExecs() && !execution(* go()) && goCut() {
      println("Intercepted message: " +
          thisJoinPointStaticPart.getSignature().getName());
      println("in class: " +
          thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
      printParameters(thisJoinPoint);
      println("Running original method: \n" );
      Object result = proceed();
      println("  result: " + result );
      return result;
   }

   static private void printParameters(JoinPoint jp) {
      println("Arguments: " );
      Object[] args = jp.getArgs();
      String[] names = ((CodeSignature)jp.getSignature()).getParameterNames();
      Class[] types = ((CodeSignature)jp.getSignature()).getParameterTypes();
      for (int i = 0; i < args.length; i++) {
         println("  "  + i + ". " + names[i] +
             " : " +            types[i].getName() +
             " = " +            args[i]);
      }
   }
}
```

#### Printing the parameters

The static portions of the parameter details, the name and types of the parameters, can be accessed through the `CodeSignature` associated with the join point. All execution join points have code signatures, so the cast to `CodeSignature` cannot fail.

The dynamic portions of the parameter details, the actual values of the parameters, are accessed directly from the execution join point object.

## Roles and Views Using Introduction

(The code for this example is in *`InstallDir`*`/examples/introduction`.)

Like advice, pieces of introduction are members of an aspect. They define new members that act as if they were defined on another class. Unlike advice, introduction affects not only the behavior of the application, but also the structural relationship between an application's classes.

This is crucial: Affecting the class structure of an application at makes these modifications available to other components of the application.

Introduction modifies a class by adding or changing

- member fields
- member methods
- nested classes

and by making the class

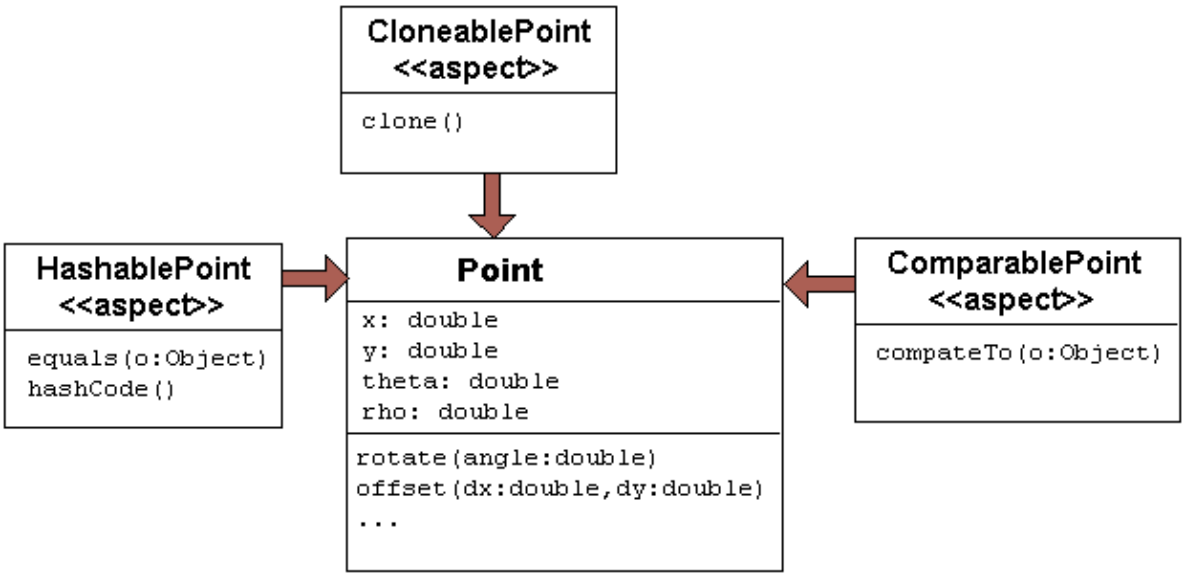
- implement interfaces
- extend classes

This example provides three illustrations of the use of introduction to encapsulate roles or views of a class. The class we will be introducing into, `Point`, is a simple class with rectangular and polar coordinates. Our introduction will make the class `Point`, in turn, cloneable, hashable, and comparable. These facilities are provided by introduction forms without having to modify the class `Point`.

### The class `Point`

The class `Point` defines geometric points whose interface includes polar and rectangular coordinates, plus some simple operations to relocate points. `Point`'s implementation has attributes for both its polar and rectangular coordinates, plus flags to indicate which currently reflect the position of the point. Some operations cause the polar coordinates to be updated from the rectangular, and some have the opposite effect. This implementation, which is in intended to give the minimum number of conversions between coordinate systems, has the property that not all the attributes stored in a `Point` object are necessary to give a canonical representation such as might be used for storing, comparing, cloning or making hash codes from points. Thus the aspects, though simple, are not totally trivial.

The diagram below gives an overview of the aspects and their interaction with the class `Point`.




### Making `Points` Cloneable  The Aspect `CloneablePoint`

This first example demonstrates the introduction of a interface (`Cloneable`) and a method (`clone`) into the class `Point`. In Java, all objects inherit the method `clone` from the class `Object`, but an object is not

cloneable unless its class also implements the interface `Cloneable`. In addition, classes frequently have requirements over and above the simple bit−for−bit copying that `Object.clone` does. In our case, we want to update a `Point`'s coordinate systems before we actually clone the `Point`. So we have to override `Object.clone` with a new method that does what we want.

The `CloneablePoint` aspect uses the `declare parents` form to introduce the interface `Cloneable` into the class `Point`. It then defines a method, `Point.clone`, which overrides the method `clone` that was inherited from `Object`. `Point.clone` updates the `Point`'s coordinate systems before invoking its superclass' `clone` method.

```
public aspect CloneablePoint {

   declare parents: Point implements Cloneable;

   public Object Point.clone() throws CloneNotSupportedException {
      // we choose to bring all fields up to date before cloning.
      makeRectangular();
      makePolar();
      return super.clone();
   }

   public static void main(String[] args){
      Point p1 = new Point();
      Point p2 = null;

      p1.setPolar(Math.PI, 1.0);
      try {
         p2 = (Point)p1.clone();
      } catch (CloneNotSupportedException e) {}
      System.out.println("p1 =" + p1 );
      System.out.println("p2 =" + p2 );

      p1.rotate(Math.PI / −2);
      System.out.println("p1 =" + p1 );
      System.out.println("p2 =" + p2 );
   }
}
```

Note that since aspects define types just as classes define types, we can define a `main` method that is invocable from the command line to use as a test method.

### Making `Points` Comparable  The Aspect `ComparablePoint`

This second example introduces another interface and method into the class `Point`.

The interface `Comparable` defines the single method `compareTo` which can be use to define a natural ordering relation among the objects of a class that implement it.

The aspect `ComparablePoint` introduces implements `Comparable` into `Point` along with a `compareTo` method that can be used to compare `Point`s. A `Point p1` is said to be less than another `Point p2` if `p1` is closer to the origin.

```
public aspect ComparablePoint {

   declare parents: Point implements Comparable;
```

```
   public int Point.compareTo(Object o) {
      return (int) (this.getRho() − ((Point)o).getRho());
   }

   public static void main(String[] args){
      Point p1 = new Point();
      Point p2 = new Point();

      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

      p1.setRectangular(2,5);
      p2.setRectangular(2,5);
      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

      p2.setRectangular(3,6);
      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

      p1.setPolar(Math.PI, 4);
      p2.setPolar(Math.PI, 4);
      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

      p1.rotate(Math.PI / 4.0);
      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

      p1.offset(1,1);
      System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
   }
}
```

### Making `Points` Hashable  The Aspect `HashablePoint`

The third aspect overrides two previously defined methods to give to `Point` the hashing behavior we want.

The method `Object.hashCode` returns an unique integer, suitable for use as a hash table key. Different implementations are allowed return different integers, but must return distinct integers for distinct objects, and the same integer for objects that test equal. But since the default implementation of `Object.equal` returns `true` only when two objects are identical, we need to redefine both `equals` and `hashCode` to work correctly with objects of type `Point`. For example, we want two `Point` objects to test equal when they have the same `x` and `y` values, or the same `rho` and `theta` values, not just when they refer to the same object. We do this by overriding the methods `equals` and `hashCode` in the class `Point`.

The class `HashablePoint` introduces the methods `hashCode` and `equals` into the class `Point`. These methods use `Point`'s rectangular coordinates to generate a hash code and to test for equality. The `x` and `y` coordinates are obtained using the appropriate get methods, which ensure the rectangular coordinates are up−to−date before returning their values.

```
public aspect HashablePoint {

   public int Point.hashCode() {
      return (int) (getX() + getY() % Integer.MAX_VALUE);
   }

   public boolean Point.equals(Object o) {
      if (o == this) { return true; }
      if (!(o instanceof Point)) { return false; }
      Point other = (Point)o;
```

```
      return (getX() == other.getX()) && (getY() == other.getY());
   }

   public static void main(String[] args) {
      Hashtable h = new Hashtable();
      Point p1 = new Point();

      p1.setRectangular(10, 10);
      Point p2 = new Point();

      p2.setRectangular(10, 10);

      System.out.println("p1 = " + p1);
      System.out.println("p2 = " + p2);
      System.out.println("p1.hashCode() = " + p1.hashCode());
      System.out.println("p2.hashCode() = " + p2.hashCode());

      h.put(p1, "P1");
      System.out.println("Got: " + h.get(p2));
   }
}
```

Again, we supply a `main` method in the aspect for testing.

# Development Aspects

## Tracing Aspects

(The code for this example is in *`InstallDir`*`/examples/tracing`.)

### Overview

Writing a class that provides tracing functionality is easy: a couple of functions, a boolean flag for turning tracing on and off, a choice for an output stream, maybe some code for formatting the output−−−these are all elements that `Trace` classes have been known to have. `Trace` classes may be highly sophisticated, too, if the task of tracing the execution of a program demands so.

But developing the support for tracing is just one part of the effort of inserting tracing into a program, and, most likely, not the biggest part. The other part of the effort is calling the tracing functions at appropriate times. In large systems, this interaction with the tracing support can be overwhelming. Plus, tracing is one of those things that slows the system down, so these calls should often be pulled out of the system before the product is shipped. For these reasons, it is not unusual for developers to write ad−hoc scripting programs that rewrite the source code by inserting/deleting trace calls before and after the method bodies.

AspectJ can be used for some of these tracing concerns in a less ad−hoc way. Tracing can be seen as a concern that crosscuts the entire system and as such is amenable to encapsulation in an aspect. In addition, it is fairly independent of what the system is doing. Therefore tracing is one of those kind of system aspects that can potentially be plugged in and unplugged without any side−effects in the basic functionality of the system.

### An Example Application

Throughout this example we will use a simple application that contains only four classes. The application is about shapes. The `TwoDShape` class is the root of the shape hierarchy:

```
public abstract class TwoDShape {
    protected double x, y;
    protected TwoDShape(double x, double y) {
        this.x = x; this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double distance(TwoDShape s) {
        double dx = Math.abs(s.getX() − x);
        double dy = Math.abs(s.getY() − y);
        return Math.sqrt(dx*dx + dy*dy);
    }
    public abstract double perimeter();
    public abstract double area();
    public String toString() {
        return (" @ (" + String.valueOf(x) + ", " + String.valueOf(y) + ") ");
    }
}
```

`TwoDShape` has two subclasses, `Circle` and `Square`:

```
public class Circle extends TwoDShape {
    protected double r;
    public Circle(double x, double y, double r) {
        super(x, y); this.r = r;
    }
    public Circle(double x, double y) { this(  x,   y, 1.0); }
    public Circle(double r)           { this(0.0, 0.0,   r); }
    public Circle()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 2 * Math.PI * r;
    }
    public double area() {
        return Math.PI * r*r;
    }
    public String toString() {
        return ("Circle radius = " + String.valueOf(r) + super.toString());
    }
}

public class Square extends TwoDShape {
    protected double s;    // side
    public Square(double x, double y, double s) {
        super(x, y); this.s = s;
    }
    public Square(double x, double y) { this(  x,   y, 1.0); }
    public Square(double s)           { this(0.0, 0.0,   s); }
    public Square()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 4 * s;
    }
    public double area() {
        return s*s;
    }
    public String toString() {
```

```
        return ("Square side = " + String.valueOf(s) + super.toString());
    }
}
```

To run this application, compile the classes. You can do it with or without ajc, the AspectJ compiler. If you've installed AspectJ, go to the directory *InstallDir*/examples and type:

```
ajc –argfile tracing/notrace.lst
```

To run the program, type

```
java tracing.ExampleMain
```

(we don't need anything special on the classpath since this is pure Java code). You should see the following output:

```
c1.perimeter() = 12.566370614359172
c1.area() = 12.566370614359172
s1.perimeter() = 4.0
s1.area() = 1.0
c2.distance(c1) = 4.242640687119285
s1.distance(c1) = 2.23606797749979
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

### Tracing Version 1

In a first attempt to insert tracing in this application, we will start by writing a `Trace` class that is exactly what we would write if we didn't have aspects. The implementation is in `version1/Trace.java`. Its public interface is:

```
public class Trace {
    public static int TRACELEVEL = 0;
    public static void initStream(PrintStream s) {...}
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}
```

If we didn't have AspectJ, we would have to insert calls to `traceEntry` and `traceExit` in all methods and constructors we wanted to trace, and to initialize `TRACELEVEL` and the stream. If we wanted to trace all the methods and constructors in our example, that would amount to around 40 calls, and we would hope we had not forgotten any method. But we can do that more consistently and reliably with the following aspect (found in `version1/TraceMyClasses.java`):

```
aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
```

```
    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

This aspect performs the tracing calls at appropriate times. According to this aspect, tracing is performed at the entrance and exit of every method and constructor defined within the shape hierarchy.

What is printed at before and after each of the traced join points is the signature of the method executing. Since the signature is static information, we can get it through `thisJoinPointStaticPart`.

To run this version of tracing, go to the directory *`InstallDir`*`/examples` and type:

```
ajc –argfile tracing/tracev1.lst
```

Running the main method of `tracing.version1.TraceMyClasses` should produce the output:

```
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.Circle(double)
  <-- tracing.Circle(double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Square(double, double, double)
  <-- tracing.Square(double, double, double)
  --> tracing.Square(double, double)
  <-- tracing.Square(double, double)
  --> double tracing.Circle.perimeter()
  <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
  --> double tracing.Circle.area()
  <-- double tracing.Circle.area()
c1.area() = 12.566370614359172
  --> double tracing.Square.perimeter()
  <-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
  --> double tracing.Square.area()
  <-- double tracing.Square.area()
s1.area() = 1.0
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
  <-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
```

```
  <-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
  --> String tracing.Square.toString()
    --> String tracing.TwoDShape.toString()
    <-- String tracing.TwoDShape.toString()
  <-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

When `TraceMyClasses.java` is not provided to **ajc**, the aspect does not have any affect on the system and the tracing is unplugged.

### Tracing Version 2

Another way to accomplish the same thing would be to write a reusable tracing aspect that can be used not only for these application classes, but for any class. One way to do this is to merge the tracing functionality of `Trace version1` with the crosscutting support of `TraceMyClasses version1`. We end up with a `Trace` aspect (found in `version2/Trace.java`) with the following public interface

```
abstract aspect Trace {

    public static int TRACELEVEL = 2;
    public static void initStream(PrintStream s) {...}
    protected static void traceEntry(String str) {...}
    protected static void traceExit(String str) {...}
    abstract pointcut myClass();
}
```

In order to use it, we need to define our own subclass that knows about our application classes, in `version2/TraceMyClasses.java`:

```
public aspect TraceMyClasses extends Trace {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);

    public static void main(String[] args) {
        Trace.TRACELEVEL = 2;
        Trace.initStream(System.err);
        ExampleMain.main(args);
    }
}
```

Notice that we've simply made the pointcut `classes`, that was an abstract pointcut in the super−aspect, concrete. To run this version of tracing, go to the directory `examples` and type:

```
  ajc −argfile tracing/tracev2.lst
```

The file tracev2.lst lists the application classes as well as this version of the files Trace.java and TraceMyClasses.java. Running the main method of `tracing.version2.TraceMyClasses` should output exactly the same trace information as that from version 1.

The entire implementation of the new `Trace` class is:

```
abstract aspect Trace {

    // implementation part
```

```
    public static int TRACELEVEL = 2;
    protected static PrintStream stream = System.err;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }
    protected static void traceEntry(String str) {
        if (TRACELEVEL == 0) return;
        if (TRACELEVEL == 2) callDepth++;
        printEntering(str);
    }
    protected static void traceExit(String str) {
        if (TRACELEVEL == 0) return;
        printExiting(str);
        if (TRACELEVEL == 2) callDepth--;
    }
    private static void printEntering(String str) {
        printIndent();
        stream.println("--> " + str);
    }
    private static void printExiting(String str) {
        printIndent();
        stream.println("<-- " + str);
    }
    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print("  ");
    }

    // protocol part

    abstract pointcut myClass();

    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before(): myConstructor() {
        traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before(): myMethod() {
        traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

This version differs from version 1 in several subtle ways. The first thing to notice is that this `Trace` class merges the functional part of tracing with the crosscutting of the tracing calls. That is, in version 1, there was a sharp separation between the tracing support (the class `Trace`) and the crosscutting usage of it (by the class `TraceMyClasses`). In this version those two things are merged. That's why the description of this class explicitly says that "Trace messages are printed before and after constructors and methods are," which is what we wanted in the first place. That is, the placement of the calls, in this version, is established by the aspect class itself, leaving less opportunity for misplacing calls.

A consequence of this is that there is no need for providing traceEntry and traceExit as public operations of this class. You can see that they were classified as protected. They are supposed to be internal implementation details of the advice.

The key piece of this aspect is the abstract pointcut classes that serves as the base for the definition of the pointcuts constructors and methods. Even though `classes` is abstract, and therefore no concrete classes are mentioned, we can put advice on it, as well as on the pointcuts that are based on it. The idea is "we don't know exactly what the pointcut will be, but when we do, here's what we want to do with it." In some ways, abstract pointcuts are similar to abstract methods. Abstract methods don't provide the implementation, but you know that the concrete subclasses will, so you can invoke those methods.

# Production Aspects

## A Bean Aspect

(The code for this example is in *`InstallDir`*`/examples/bean`.)

This example examines an aspect that makes Point objects into a Java beans with bound properties.

### Introduction

Java beans are reusable software components that can be visually manipulated in a builder tool. The requirements for an object to be a bean are few. Beans must define a no−argument constructor and must be either `Serializable` or `Externalizable`. Any properties of the object that are to be treated as bean properties should be indicated by the presence of appropriate `get` and `set` methods whose names are `get`*property* and `set` *property* where *property* is the name of a field in the bean class. Some bean properties, known as bound properties, fire events whenever their values change so that any registered listeners (such as, other beans) will be informed of those changes. Making a bound property involves keeping a list of registered listeners, and creating and dispatching event objects in methods that change the property values, such as set*property* methods.

`Point` is a simple class representing points with rectangular coordinates. `Point` does not know anything about being a bean: there are set methods for `x` and `y` but they do not fire events, and the class is not serializable. Bound is an aspect that makes `Point` a serializable class and makes its `get` and `set` methods support the bound property protocol.

### The Class `Point`

The class `Point` is a very simple class with trivial getters and setters, and a simple vector offset method.

```
class Point {

  protected int x = 0;
  protected int y = 0;

  public int getX() {
    return x;
```

```
  }

  public int getY() {
    return y;
  }

  public void setRectangular(int newX, int newY) {
    setX(newX);
    setY(newY);
  }

  public void setX(int newX) {
    x = newX;
  }

  public void setY(int newY) {
    y = newY;
  }

  public void offset(int deltaX, int deltaY) {
    setRectangular(x + deltaX, y + deltaY);
  }

  public String toString() {
    return "(" + getX() + ", " + getY() + ")" ;
  }
}
```

### The Aspect `BoundPoint`

The aspect `BoundPoint` adds "beanness" to `Point` objects. The first thing it does is privately introduce a reference to an instance of `PropertyChangeSupport` into all `Point` objects. The property change support object must be constructed with a reference to the bean for which it is providing support, so it is initialized by passing it this, an instance of `Point`. The support field is privately introduced, so only the code in the aspect can refer to it.

Methods for registering and managing listeners for property change events are introduced into `Point` by the introductions. These methods delegate the work to the property change support object.

The introduction also makes `Point` implement the `Serializable` interface. Implementing `Serializable` does not require any methods to be implemented. Serialization for `Point` objects is provided by the default serialization method.

The pointcut `setters` names the `set` methods: reception by a `Point` object of any method whose name begins with '`set`' and takes one parameter. The around advice on `setters()` stores the values of the `X` and `Y` properties, calls the original `set` method and then fires the appropriate property change event according to which set method was called. Note that the call to the method proceed needs to pass along the `Point p`. The rule of thumb is that context that an around advice exposes must be passed forward to continue.

```
aspect BoundPoint {
  private PropertyChangeSupport Point.support = new PropertyChangeSupport(this);

  public void Point.addPropertyChangeListener(PropertyChangeListener listener){
    support.addPropertyChangeListener(listener);
  }
```

```
  public void Point.addPropertyChangeListener(String propertyName,
                                              PropertyChangeListener listener){

    support.addPropertyChangeListener(propertyName, listener);
  }

  public void Point.removePropertyChangeListener(String propertyName,
                                                 PropertyChangeListener listener) {
    support.removePropertyChangeListener(propertyName, listener);
  }

  public void Point.removePropertyChangeListener(PropertyChangeListener listener) {
    support.removePropertyChangeListener(listener);
  }

  public void Point.hasListeners(String propertyName) {
    support.hasListeners(propertyName);
  }

  declare parents: Point implements Serializable;

  pointcut setter(Point p): call(void Point.set*(*)) && target(p);

  void around(Point p): setter(p) {
        String propertyName =
      thisJoinPointStaticPart.getSignature().getName().substring("set".length());
        int oldX = p.getX();
        int oldY = p.getY();
        proceed(p);
        if (propertyName.equals("X")){
      firePropertyChange(p, propertyName, oldX, p.getX());
        } else {
      firePropertyChange(p, propertyName, oldY, p.getY());
        }
  }

  void firePropertyChange(Point p,
                          String property,
                          double oldval,
                          double newval) {
        p.support.firePropertyChange(property,
                                     new Double(oldval),
                                     new Double(newval));
  }
}
```

### The Test Program

The test program registers itself as a property change listener to a `Point` object that it creates and then performs simple manipulation of that point: calling its set methods and the offset method. Then it serializes the point and writes it to a file and then reads it back. The result of saving and restoring the point is that a new point is created.

```
  class Demo implements PropertyChangeListener {

    static final String fileName = "test.tmp";

    public void propertyChange(PropertyChangeEvent e){
      System.out.println("Property " + e.getPropertyName() + " changed from " +
```

```
        e.getOldValue() + " to " + e.getNewValue() );
    }

    public static void main(String[] args){
      Point p1 = new Point();
      p1.addPropertyChangeListener(new Demo());
      System.out.println("p1 =" + p1);
      p1.setRectangular(5,2);
      System.out.println("p1 =" + p1);
      p1.setX( 6 );
      p1.setY( 3 );
      System.out.println("p1 =" + p1);
      p1.offset(6,4);
      System.out.println("p1 =" + p1);
      save(p1, fileName);
      Point p2 = (Point) restore(fileName);
      System.out.println("Had: " + p1);
      System.out.println("Got: " + p2);
      }
    ...
  }
```

### Compiling and Running the Example

To compile and run this example, go to the examples directory and type:

```
ajc −argfile bean/files.lst
java bean.Demo
```

## The Subject/Observer Protocol

(The code for this example is in *`InstallDir`*`/examples/observer`.)

This demo illustrates how the Subject/Observer design pattern can be coded with aspects.

### Overview

The demo consists of the following: A colored label is a renderable object that has a color that cycles through a set of colors, and a number that records the number of cycles it has been through. A button is an action item that records when it is clicked.

With these two kinds of objects, we can build up a Subject/Observer relationship in which colored labels observe the clicks of buttons; that is, where colored labels are the observers and buttons are the subjects.

The demo is designed and implemented using the Subject/Observer design pattern. The remainder of this example explains the classes and aspects of this demo, and tells you how to run it.

### Generic Components

The generic parts of the protocol are the interfaces `Subject` and `Observer`, and the abstract aspect `SubjectObserverProtocol`. The `Subject` interface is simple, containing methods to add, remove,

and view `Observer` objects, and a method for getting data about state changes:

```
   interface Subject {
     void addObserver(Observer obs);
     void removeObserver(Observer obs);
     Vector getObservers();
     Object getData();
 }
```

The `Observer` interface is just as simple, with methods to set and get `Subject` objects, and a method to call when the subject gets updated.

```
 interface Observer {
     void setSubject(Subject s);
     Subject getSubject();
     void update();
 }
```

The `SubjectObserverProtocol` aspect contains within it all of the generic parts of the protocol, namely, how to fire the `Observer` objects' update methods when some state changes in a subject.

```
 abstract aspect SubjectObserverProtocol {

     abstract pointcut stateChanges(Subject s);

     after(Subject s): stateChanges(s) {
         for (int i = 0; i < s.getObservers().size(); i++) {
             ((Observer)s.getObservers().elementAt(i)).update();
         }
     }

     private Vector Subject.observers = new Vector();
     public void   Subject.addObserver(Observer obs) {
         observers.addElement(obs);
         obs.setSubject(this);
     }
     public void   Subject.removeObserver(Observer obs) {
         observers.removeElement(obs);
         obs.setSubject(null);
     }
     public Vector Subject.getObservers() { return observers; }

     private Subject Observer.subject = null;
     public void     Observer.setSubject(Subject s) { subject = s; }
     public Subject  Observer.getSubject() { return subject; }

 }
```

Note that this aspect does three things. It define an abstract pointcut that extending aspects can override. It defines advice that should run after the join points of the pointcut. And it introduces state and behavior onto the `Subject` and `Observer` interfaces.

### Application Classes

`Button` objects extend `java.awt.Button`, and all they do is make sure the `void click()` method is called whenever a button is clicked.

```
  class Button extends java.awt.Button {

      static final Color  defaultBackgroundColor = Color.gray;
      static final Color  defaultForegroundColor = Color.black;
      static final String defaultText = "cycle color";

      Button(Display display) {
          super();
          setLabel(defaultText);
          setBackground(defaultBackgroundColor);
          setForeground(defaultForegroundColor);
          addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                      Button.this.click();
                  }
              });
          display.addToFrame(this);
      }

      public void click() {}

  }
```

Note that this class knows nothing about being a Subject.

ColorLabel objects are labels that support the void colorCycle() method. Again, they know nothing about being an observer.

```
  class ColorLabel extends Label {

      ColorLabel(Display display) {
          super();
          display.addToFrame(this);
      }

      final static Color[] colors = {Color.red, Color.blue,
                                     Color.green, Color.magenta};
      private int colorIndex = 0;
      private int cycleCount = 0;
      void colorCycle() {
          cycleCount++;
          colorIndex = (colorIndex + 1) % colors.length;
          setBackground(colors[colorIndex]);
          setText("" + cycleCount);
      }
  }
```

Finally, the `SubjectObserverProtocolImpl` implements the subject/observer protocol, with `Button` objects as subjects and `ColorLabel` objects as observers:

```
package observer;

import java.util.Vector;

aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
    public Object Button.getData() { return this; }

    declare parents: ColorLabel implements Observer;
```

```
    public void    ColorLabel.update() {
        colorCycle();
    }

    pointcut stateChanges(Subject s):
        target(s) &&
        call(void Button.click());

}
```

It does this by introducing the appropriate interfaces onto the `Button` and `ColorLabel` classes, making sure the methods required by the interfaces are implemented, and providing a definition for the `stateChanges` pointcut. Now, every time a `Button` is clicked, all `ColorLabel` objects observing that button will `colorCycle`.

### Compiling and Running

`Demo` is the top class that starts this demo. It instantiates a two buttons and three observers and links them together as subjects and observers. So to run the demo, go to the `examples` directory and type:

```
ajc −argfile observer/files.lst
java observer.Demo
```

## A Simple Telecom Simulation

(The code for this example is in *`InstallDir`*`/examples/telecom`.)

This example illustrates some ways that dependent concerns can be encoded with aspects. It uses an example system comprising a simple model of telephone connections to which timing and billing features are added using aspects, where the billing feature depends upon the timing feature.

### The Application

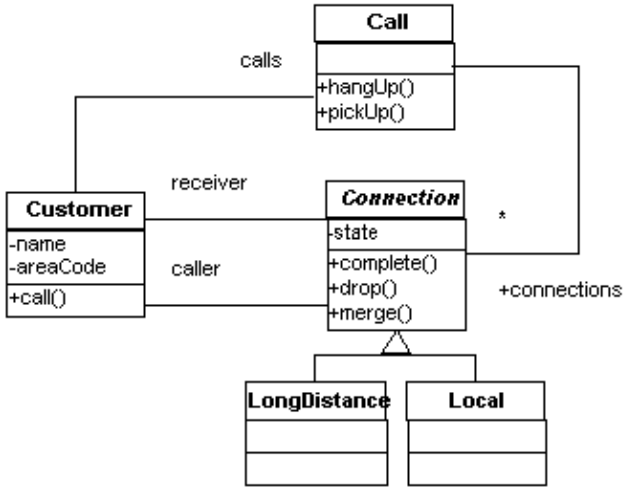
The example application is a simple simulation of a telephony system in which customers make, accept, merge and hang−up both local and long distance calls. The application architecture is in three layers.

- The basic objects provide basic functionality to simulate customers, calls and connections (regular calls have one connection, conference calls have more than one).
- The timing feature is concerned with timing the connections and keeping the total connection time per customer. Aspects are used to add a timer to each connection and to manage the total time per customer.
- The billing feature is concerned with charging customers for the calls they make. Aspects are used to calculate a charge per connection and, upon termination of a connection, to add the charge to the appropriate customer's bill. The billing aspect builds upon the timing aspect: it uses a pointcut defined in Timing and it uses the timers that are associated with connections.

The simulation of system has three configurations: basic, timing and billing. Programs for the three configurations are in classes `BasicSimulation`, `TimingSimulation` and `BillingSimulation`. These share a common superclass `AbstractSimulation`, which defines the method run with the simulation itself and the method wait used to simulate elapsed time.

### The Basic Objects

The telecom simulation comprises the classes `Customer`, `Call` and the abstract class `Connection` with its two concrete subclasses `Local` and `LongDistance`. Customers have a name and a numeric area code. They also have methods for managing calls. Simple calls are made between one customer (the caller) and another (the receiver), a `Connection` object is used to connect them. Conference calls between more than two customers will involve more than one connection. A customer may be involved in many calls at one time.




### The Class `Customer`

`Customer` has methods `call`, `pickup`, `hangup` and `merge` for managing calls.

```
public class Customer {

      private String name;
      private int areacode;
      private Vector calls = new Vector();

      protected void removeCall(Call c){
          calls.removeElement(c);
      }

      protected void addCall(Call c){
          calls.addElement(c);
      }

      public Customer(String name, int areacode) {
          this.name = name;
          this.areacode = areacode;
      }

      public String toString() {
          return name + "(" + areacode + ")";
```

```
    }

    public int getAreacode(){
        return areacode;
    }

    public boolean localTo(Customer other){
        return areacode == other.areacode;
    }

    public Call call(Customer receiver) {
        Call call = new Call(this, receiver);
        addCall(call);
        return call;
    }

    public void pickup(Call call) {
        call.pickup();
        addCall(call);
    }

    public void hangup(Call call) {
        call.hangup(this);
        removeCall(call);
    }

    public void merge(Call call1, Call call2){
        call1.merge(call2);
        removeCall(call2);
    }
}
```

### The Class `Call`

Calls are created with a caller and receiver who are customers. If the caller and receiver have the same area code then the call can be established with a `Local` connection (see below), otherwise a `LongDistance` connection is required. A call comprises a number of connections between customers. Initially there is only the connection between the caller and receiver but additional connections can be added if calls are merged to form conference calls.

### The Class `Connection`

The class `Connection` models the physical details of establishing a connection between customers. It does this with a simple state machine (connections are initially `PENDING`, then `COMPLETED` and finally `DROPPED`). Messages are printed to the console so that the state of connections can be observed. Connection is an abstract class with two concrete subclasses: `Local` and `LongDistance`.

```
abstract class Connection {

    public static final int PENDING = 0;
    public static final int COMPLETE = 1;
    public static final int DROPPED = 2;

    Customer caller, receiver;
    private int state = PENDING;
```

```
    Connection(Customer a, Customer b) {
        this.caller = a;
        this.receiver = b;
    }

    public int getState(){
        return state;
    }

    public Customer getCaller() { return caller; }

    public Customer getReceiver() { return receiver; }

    void complete() {
        state = COMPLETE;
        System.out.println("connection completed");
    }

    void drop() {
        state = DROPPED;
        System.out.println("connection dropped");
    }

    public boolean connects(Customer c){
        return (caller == c || receiver == c);
    }

}
```

### The Class Local

```
class Local extends Connection {
    Local(Customer a, Customer b) {
        super(a, b);
        System.out.println("[new local connection from " +
           a + " to " + b + "]");
    }
}
```

### The Class LongDistance

```
class LongDistance extends Connection {
    LongDistance(Customer a, Customer b) {
        super(a, b);
        System.out.println("[new long distance connection from " +
            a + " to " + b + "]");
    }
}
```

### Compiling and Running the Basic Simulation

The source files for the basic system are listed in the file `basic.lst`. To build and run the basic system, in a shell window, type these commands:

```
ajc −argfile telecom/basic.lst
java telecom.BasicSimulation
```

## Timing

The `Timing` aspect keeps track of total connection time for each `Customer` by starting and stopping a timer associated with each connection. It uses some helper classes:

#### The Class `Timer`

A `Timer` object simply records the current time when it is started and stopped, and returns their difference when asked for the elapsed time. The aspect `TimerLog` (below) can be used to cause the start and stop times to be printed to standard output.

```
class Timer {
    long startTime, stopTime;

    public void start() {
        startTime = System.currentTimeMillis();
        stopTime = startTime;
    }

    public void stop() {
        stopTime = System.currentTimeMillis();
    }

    public long getTime() {
        return stopTime − startTime;
    }
}
```

## The Aspect `TimerLog`

The aspect `TimerLog` can be included in a build to get the timer to announce when it is started and stopped.

```
public aspect TimerLog {

    after(Timer t): target(t) && call(* Timer.start())  {
      System.err.println("Timer started: " + t.startTime);
    }

    after(Timer t): target(t) && call(* Timer.stop()) {
      System.err.println("Timer stopped: " + t.stopTime);
    }
}
```

## The Aspect `Timing`

The aspect `Timing` introduces attribute `totalConnectTime` into the class `Customer` to store the accumulated connection time per `Customer`. It introduces attribute timer into `Connection` to associate a timer with each `Connection`. Two pieces of after advice ensure that the timer is started when a connection is completed and and stopped when it is dropped. The pointcut `endTiming` is defined so that it can be used by the `Billing` aspect.

```
public aspect Timing {
```

```
    public long Customer.totalConnectTime = 0;

    public long getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }
    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) { return conn.timer; }

    after (Connection c): target(c) && call(void Connection.complete()) {
        getTimer(c).start();
    }

    pointcut endTiming(Connection c): target(c) &&
        call(void Connection.drop());

    after(Connection c): endTiming(c) {
        getTimer(c).stop();
        c.getCaller().totalConnectTime += getTimer(c).getTime();
        c.getReceiver().totalConnectTime += getTimer(c).getTime();
    }
}
```

## Billing

The Billing system adds billing functionality to the telecom application on top of timing.

#### The Aspect `Billing`

The aspect `Billing` introduces attribute `payer` into `Connection` to indicate who initiated the call and therefore who is responsible to pay for it. It also introduces method `callRate` into `Connection` so that local and long distance calls can be charged differently. The call charge must be calculated after the timer is stopped; the after advice on pointcut `Timing.endTiming` does this and `Billing` dominates Timing to make sure that this advice runs after `Timing's` advice on the same join point. It introduces attribute `totalCharge` and its associated methods into `Customer` (to manage the customer's bill information.

```
public aspect Billing dominates Timing {
    // domination required to get advice on endtiming in the right order

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;


    public Customer Connection.payer;
    public Customer getPayer(Connection conn) { return conn.payer; }

    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) {
        conn.payer = cust;
    }

    public abstract long Connection.callRate();


    public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
    public long Local.callRate() { return LOCAL_RATE; }
```

```
    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }


    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) { return cust.totalCharge; }

    public void Customer.addCharge(long charge){
        totalCharge += charge;
    }
}
```

### Accessing the Introduced State

Both the aspects `Timing` and `Billing` contain the definition of operations that the rest of the system may want to access. For example, when running the simulation with one or both aspects, we want to find out how much time each customer spent on the telephone and how big their bill is. That information is also stored in the classes, but they are accessed through static methods of the aspects, since the state they refer to is private to the aspect.

Take a look at the file `TimingSimulation.java`. The most important method of this class is the method `report(Customer c)`, which is used in the method run of the superclass `AbstractSimulation`. This method is intended to print out the status of the customer, with respect to the `Timing` feature.

```
protected void report(Customer c){
    Timing t = Timing.aspectOf();
    System.out.println(c + " spent " + t.getTotalConnectTime(c));
}
```

### Compiling and Running

The files timing.lst and billing.lst contain file lists for the timing and billing configurations. To build and run the application with only the timing feature, go to the directory examples and type:

```
ajc –argfile telecom/timing.lst
java telecom.TimingSimulation
```

To build and run the application with the timing and billing features, go to the directory examples and type:

```
ajc –argfile telecom/billing.lst
java telecom.BillingSimulation
```

### Discussion

There are some explicit dependencies between the aspects Billing and Timing:

- 

Billing is declared to dominate Timing so that Billing's after advice runs after that of Timing when they are on the same join point.

- Billing uses the pointcut Timing.endTiming.
- Billing needs access to the timer associated with a connection.

# Reusable Aspects

## Tracing Aspects Revisited

(The code for this example is in *InstallDir*/examples/tracing.)

### Tracing Version 3

One advantage of not exposing the methods traceEntry and traceExit as public operations is that we can easily change their interface without any dramatic consequences in the rest of the code.

Consider, again, the program without AspectJ. Suppose, for example, that at some point later the requirements for tracing change, stating that the trace messages should always include the string representation of the object whose methods are being traced. This can be achieved in at least two ways. One way is keep the interface of the methods `traceEntry` and `traceExit` as it was before,

```
public static void traceEntry(String str);
public static void traceExit(String str);
```

In this case, the caller is responsible for ensuring that the string representation of the object is part of the string given as argument. So, calls must look like:

```
Trace.traceEntry("Square.distance in " + toString());
```

Another way is to enforce the requirement with a second argument in the trace operations, e.g.

```
public static void traceEntry(String str, Object obj);
public static void traceExit(String str, Object obj);
```

In this case, the caller is still responsible for sending the right object, but at least there is some guarantees that some object will be passed. The calls will look like:

```
Trace.traceEntry("Square.distance", this);
```

In either case, this change to the requirements of tracing will have dramatic consequences in the rest of the code −− every call to the trace operations traceEntry and traceExit must be changed!

Here's another advantage of doing tracing with an aspect. We've already seen that in version 2 `traceEntry` and `traceExit` are not publicly exposed. So changing their interfaces, or the way they are used, has only a

small effect inside the `Trace` class. Here's a partial view at the implementation of `Trace`, version 3. The differences with respect to version 2 are stressed in the comments:

```
abstract aspect Trace {

    public static int TRACELEVEL = 0;
    protected static PrintStream stream = null;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }

    protected static void traceEntry(String str, Object o) {
        if (TRACELEVEL == 0) return;
        if (TRACELEVEL == 2) callDepth++;
        printEntering(str + ": " + o.toString());
    }

    protected static void traceExit(String str, Object o) {
        if (TRACELEVEL == 0) return;
        printExiting(str + ": " + o.toString());
        if (TRACELEVEL == 2) callDepth--;
    }

    private static void printEntering(String str) {
        printIndent();
        stream.println("Entering " + str);
    }

    private static void printExiting(String str) {
        printIndent();
        stream.println("Exiting " + str);
    }


    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print("  ");
    }


    abstract pointcut myClass(Object obj);

    pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
    pointcut myMethod(Object obj): myClass(obj) &&
        execution(* *(..)) && !execution(String toString());

    before(Object obj): myConstructor(obj) {
        traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
    }
    after(Object obj): myConstructor(obj) {
        traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
    }

    before(Object obj): myMethod(obj) {
        traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
    }
    after(Object obj): myMethod(obj) {
        traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
    }
```

```
}
```

As you can see, we decided to apply the first design by preserving the interface of the methods `traceEntry` and `traceExit`. But it doesn't matter we could as easily have applied the second design (the code in the directory `examples/tracing/version3` has the second design). The point is that the effects of this change in the tracing requirements are limited to the `Trace` aspect class.

One implementation change worth noticing is the specification of the pointcuts. They now expose the object. To maintain full consistency with the behavior of version 2, we should have included tracing for static methods, by defining another pointcut for static methods and advising it. We leave that as an exercise.

Moreover, we had to exclude the method `toString` from the `methods` pointcut. The problem here is that `toString` is being called from inside the advice. Therefore if we trace it, we will end up in an infinite recursion of calls. This is a subtle point, and one that you must be aware when writing advice. If the advice calls back to the objects, there is always the possibility of recursion. Keep that in mind!

In summary, to implement the change in the tracing requirements we had to make a couple of changes in the implementation of the `Trace` aspect class, including changing the specification of the pointcuts. That's only natural. But the implementation changes were limited to this aspect. Without aspects, we would have to change the implementation of every application class.

Finally, to run this version of tracing, go to the directory `examples` and type:

```
ajc –argfile tracing/tracev3.lst
```

The file tracev3.lst lists the application classes as well as this version of the files `Trace.java` and `TraceMyClasses.java`. To run the program, type

```
java tracing.version3.TraceMyClasses
```

The output should be:

```
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.Circle(double)
  <-- tracing.Circle(double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Square(double, double, double)
  <-- tracing.Square(double, double, double)
  --> tracing.Square(double, double)
  <-- tracing.Square(double, double)
  --> double tracing.Circle.perimeter()
  <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
  --> double tracing.Circle.area()
  <-- double tracing.Circle.area()
c1.area() = 12.566370614359172
  --> double tracing.Square.perimeter()
  <-- double tracing.Square.perimeter()
```

```
s1.perimeter() = 4.0
  --> double tracing.Square.area()
  <-- double tracing.Square.area()
s1.area() = 1.0
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
  <-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
  <-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
  --> String tracing.Square.toString()
    --> String tracing.TwoDShape.toString()
    <-- String tracing.TwoDShape.toString()
  <-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

# Chapter 4. Pitfalls

**Table of Contents**



## About this Chapter

This chapter consists of aspectj programs that may lead to surprising behaviour and how to understand them.

## Infinite loops

Here is a Java program with peculiar behavior

```
public class Main {
    public static void main(String[] args) {
        foo();
        System.out.println("done with call to foo");
    }

    static void foo() {
        try {
            foo();
        } finally {
            foo();
        }
    }
}
```

This program will never reach the println call, but when it aborts will have no stack trace.

This silence is caused by multiple StackOverflowExceptions. First the infinite loop in the body of the method generates one, which the finally clause tries to handle. But this finally clause also generates an infinite loop which the current JVMs can't handle gracefully leading to the completely silent abort.

The following short aspect will also generate this behavior:

```
aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after():  call(* *(..)) { System.out.println("after"); }
}
```

Why? Because the call to println is also a call matched by the pointcut `call (* *(..))`. We get no output because we used simple after() advice. If the aspect were changed to

```
aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after() returning:  call(* *(..)) { System.out.println("after"); }
}
```

Then at least a StackOverflowException with a stack trace would be seen. In both cases, though, the overall problem is advice applying within its own body.

There's a simple idiom to use if you ever have a worry that your advice might apply in this way. Just restrict the advice from occurring in join points caused within the aspect. So:

```
aspect A {
    before(): call(* *(..)) && !within(A) { System.out.println("before"); }
    after() returning:  call(* *(..)) && !within(A) { System.out.println("after"); }
}
```

Other solutions might be to more closely restrict the pointcut in other ways, for example:

```
aspect A {
    before(): call(* MyObject.*(..))  { System.out.println("before"); }
    after() returning:  call(* MyObject.*(..))  { System.out.println("after"); }
}
```

The moral of the story is that unrestricted generic pointcuts can pick out more join points than intended.

# Appendix A. AspectJ Quick Reference

**Table of Contents**

# Pointcut Designators

**Table A.1. Pointcut Designators**

| **Methods and Constructors** | |
|---|---|
| `call(`*`Signature`*`)` | Method or constructor call join points when the signature matches *`Signature`* |
| `execution(`*`Signature`*`)` | Method or constructor execution join points when the signature matches *`Signature`* |
| `initialization(`*`Signature`*`)` | Object initialization join point when the first constructor called in the type matches *`Signature`* |
| **Exception Handlers** | |
| `handler(`*`TypePattern`*`)` | Exception handler execution join points when try handlers for the throwable types in *`TypePattern`* are executed. The exception object can be accessed with an `args` pointcut. |
| **Fields** | |
| `get(`*`Signature`*`)` | Field reference join points when the field matches *`Signature`* |
| `set(`*`Signature`*`)` | Field assignment join points when the field matches *`Signature`*. The new value can be accessed with an `args` pointcut. |
| **Static Initializers** | |
| `staticinitialization(`*`TypePattern`*`)` | Static initializer execution join points for the types in *`TypePattern`*. |
| **Objects** | |
| `this(`*`TypePattern`*`)` | Join points when the currently executing object is an instance of a type in *`TypePattern`* |
| `target(`*`TypePattern`*`)` | Join points when the target object is an instance of a type in *`TypePattern`* |
| `args(`*`TypePattern`*`, ...)` | Join points when the argument objects are instances of the *`TypePattern`*s |
| **Lexical Extents** | |
| `within(`*`TypePattern`*`)` | Join points when the code executing is defined in the types in *`TypePattern`* |
| `withincode(`*`Signature`*`)` | Join points when the code executing is defined in the method or constructor with signature *`Signature`* |
| **Control Flow** | |
| `cflow(`*`Pointcut`*`)` | Join points in the control flow of the join points specified by *`Pointcut`* |
| `cflowbelow(`*`Pointcut`*`)` | Join points in the control flow below the join points specified by *`Pointcut`* |
| **Conditional** | |
| `if(`*`Expression`*`)` | Join points when the boolean *`Expression`* evaluates to `true` |
| **Combination** | |
| `!` *`Pointcut`* | Join points that are not picked out by *`Pointcut`* |
| *`Pointcut0`* `&&` *`Pointcut1`* | Join points that are picked out by both *`Pointcut0`* and *`Pointcut1`* |
| *`Pointcut0`* `||` *`Pointcut1`* | Join points that are picked out by either *`Pointcut0`* or *`Pointcut1`* |
| `(` *`Pointcut`* `)` | Join points that are picked out by the parenthesized *`Pointcut`* |

# Type Patterns

**Table A.2. Type Name Patterns**

| | |
|---|---|
| * alone | all types |
| * in an identifier | any sequence of characters, not including "." |
| .. in an identifier | any sequence of characters starting and ending with "." |

The + wildcard can be appended to a type name pattern to indicate all subtypes.

Any number of []s can be put on a type name or subtype pattern to indicate array types.

**Table A.3. Type Patterns**

| | |
|---|---|
| *TypeNamePattern* | all types in *TypeNamePattern* |
| *SubtypePattern* | all types in *SubtypePattern*, a pattern with a +. |
| *ArrayTypePattern* | all types in *ArrayTypePattern*, a pattern with one or more []s. |
| !*TypePattern* | all types not in *TypePattern* |
| *TypePattern0* && *TypePattern1* | all types in both *TypePattern0* and *TypePattern1* |
| *TypePattern0* \|\| *TypePattern1* | all types in either *TypePattern0* or *TypePattern1* |
| ( *TypePattern* ) | all types in *TypePattern* |

# Advice

**Table A.4. Advice**

| | |
|---|---|
| before(*Formals*) : | Run before the join point. |
| after(*Formals*) returning [ (*Formal*) ] : | Run after the join point if it returns normally. The optional formal gives access to the returned value. |
| after(*Formals*) throwing [ (*Formal*) ] : | Run after the join point if it throws an exception. The optional formal gives access to the Throwable exception value. |
| after(*Formals*) : | Run after the join point both when it returns normally and when it throws an exception. |
| *Type* around(*Formals*) [ throws *TypeList* ] : | Run instead of the join point. The join point can be executed by calling proceed. |

# Static Crosscutting

**Table A.5. Introduction**

| | |
|---|---|
| *Modifiers Type TypePattern.Id(Formals) { Body }* | Defines a method on the types in *TypePattern*. |
| abstract *Modifiers Type TypePattern.Id(Formals);* | Defines an abstract method on the types in *TypePattern*. |
| *Modifiers TypePattern*.new*(Formals) { Body }* | Defines a a constructor on the types in *TypePattern*. |

| | |
|---|---|
| *Modifiers Type TypePattern.Id [ = Expression ];* | Defines a field on the types in *TypePattern*. |

**Table A.6. Other declarations**

| | |
|---|---|
| `declare parents: TypePattern extends TypeList;` | Declares that the types in *TypePattern* extend the types of *TypeList*. |
| `declare parents: TypePattern implements TypeList;` | Declares that the types in *TypePattern* implement the types of *TypeList*. |
| `declare warning: Pointcut: String;` | Declares that if any of the join points in *Pointcut* possibly exist in the program, the compiler should emit a warning of *String*. |
| `declare error: Pointcut: String;` | Declares that if any of the join points in *Pointcut* possibly exist in the program, the compiler should emit an error of *String*. |
| `declare soft: TypePattern: Pointcut;` | Declares that any exception of a type in *TypePattern* that gets thrown at any join point picked out by *Pointcut* will be wrapped in `org.aspectj.lang.SoftException`. |

# Aspect Associations

**Table A.7. Associations**

| modifier | Description | Accessor |
|---|---|---|
| [ `issingleton` ] | One instance of the aspect is made. This is the default. | `aspectOf()` at all join points |
| `perthis(Pointcut)` | An instance is associated with each object that is the currently executing object at any join point in *Pointcut*. | `aspectOf(Object)` at all join points |
| `pertarget(Pointcut)` | An instance is associated with each object that is the target object at any join point in *Pointcut*. | `aspectOf(Object)` at all join points |
| `percflow(Pointcut)` | The aspect is defined for each entrance to the control flow of the join points defined by *Pointcut*. | `aspectOf()` at join points in `cflow(Pointcut)` |
| `percflowbelow(Pointcut)` | The aspect is defined for each entrance to the control flow below the join points defined by *Pointcut*. | `aspectOf()` at join points in `cflowbelow(Pointcut)` |

# Appendix B. Language Semantics

**Table of Contents**

AspectJ extends Java by overlaying a concept of join points onto the existing Java semantics and by adding adds four kinds of program elements to Java:

Join points are well−defined points in the execution of a program. These include method and constructor calls, field accesses and others described below.

A pointcut picks out join points, and exposes some of the values in the execution context of those join points. There are several primitive pointcut designators, new named pointcuts can be defined by the `pointcut` declaration.

Advice is code that executes at each join point in a pointcut. Advice has access to the values exposed by the pointcut. Advice is defined by `before`, `after`, and `around` declarations.

Introduction and declaration form AspectJ's static crosscutting features, that is, is code that may change the type structure of a program, by adding to or extending interfaces and classes with new fields, constructors, or methods. Introductions are defined through an extension of usual method, field, and constructor declarations, and other declarations are made with a new `declare` keyword.

An aspect is a crosscutting type, that encapsulates pointcuts, advice, and static crosscutting features. By type, we mean Java's notion: a modular unit of code, with a well−defined interface, about which it is possible to do reasoning at compile time. Aspects are defined by the `aspect` declaration.

# Join Points

While aspects do define crosscutting types, the AspectJ system does not allow completely arbitrary crosscutting. Rather, aspects define types that cut across principled points in a program's execution. These principled points are called join points.

A join point is a well−defined point in the execution of a program. The join points defined by AspectJ are:

*Method call*

When a method is called, not including super calls.

*Method execution*
When the body of code for an actual method executes.

*Constructor call*
When an object is built and a constructor is called, not including this or super constructor calls.

*Initializer execution*
When the non−static initializers of a class run.

*Constructor execution*
When the body of code for an actual constructor executes, after its this or super constructor call.

*Static initializer execution*
When the static initializer for a class executes.

*Object pre−initialization*
Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. Thus, the execution of these join points encompass the join points from the code found in `this()` and `super()` constructor calls.

*Object initialization*
When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object.

*Field reference*
When a non−final field is referenced.

*Field assignment*
When a field is assigned to.

*Handler execution*
When an exception handler executes.

# Pointcuts

A pointcut is a program element that picks out join points, as well as data from the execution context of the join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other pointcuts. So a pointcut is defined by one of

`call(Signature)`
Picks out a method or constructor call join point based on the static signature.

`execution(Signature)`
Picks out a method or constructor execution join point based on the static signature.

*get(Signature)*
: Picks out a field get join point based on the static signature.

*set(Signature)*
: Picks out a field set join point based on the static signature.

*handler(TypePattern)*
: Picks out an exception handler of any of the Throwable types of the type pattern.

*initialization(Signature)*
: Picks out an object initialization join point based on the static signature of the starting constructor.

*staticinitialization(TypePattern)*
: Picks out a static initializer execution join point of any of the types of the type pattern.

*within(TypePattern)*
: Picks out all join points where the executing code is defined in any of the classes of the type pattern.

*withincode(Signature)*
: Picks out all join points where the executing code is defined in the method or constructor of the appropriate signature.

*cflow(Pointcut)*
: Picks out all join points in the control flow of the join points picked out by the pointcut, including pointcut's join points themselves.

*cflowbelow(Pointcut)*
: Picks out all join points in the control flow below the join points picked out by the pointcut.

*this(TypePattern or Id)*
: Picks out all join points where the currently executing object (the object bound to `this`) is an instance of a type of the type pattern, or of the type of the identifier. Will not match any join points from static methods.

*target(TypePattern or Id)*
: Picks out all join points where the target object (the object on which a call or field operation is applied to) is an instance of a type of the type pattern, or of the type of the identifier. Will not match any calls, gets, or sets to static members.

*args(TypePattern or Id, ...)*
: Picks out all join points where the arguments are instances of a type of the appropriate type pattern or identifier.

*PointcutId(TypePattern or Id, ...)*
: Picks out all join points that are picked out by the user−defined pointcut designator named by *PointcutId*.

*if(BooleanExpression)*
: Picks out all join points where the boolean expression evaluates to `true`.

*! Pointcut*

Picks out all join points that are not picked out by the pointcut.

*`Pointcut0 && Pointcut1`*
Picks out all join points that are picked out by both of the pointcuts.

*`Pointcut0 || Pointcut1`*
Picks out all join points that are picked out by either of the pointcuts.

*`( Pointcut )`*
Picks out all join points that are picked out by the parenthesized pointcut.

## Pointcut naming

A named pointcut is defined with the `pointcut` declaration.

```
pointcut publicIntCall(int i):
    call(public * *(int)) && args(i);
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as `public` or `private`.

```
class C {
    pointcut publicCall(int i):
        call(public * *(int)) && args(i);
}

class D {
    pointcut myPublicCall(int i):
        C.publicCall(i) && within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects.

```
abstract aspect A {
    abstract pointcut publicCall(int i);
}
```

In such a case, an extending aspect may override the abstract pointcut.

```
aspect B extends A {
    pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}
```

For completeness, a pointcut with a declaration may be declared `final`.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.

The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class *body*. This means that the following code is legal:

```
aspect B percflow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}
```

## Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. For example, the PublicIntCall above exposes the first argument from the receptions of all public unary integer methods. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching.

On the right−hand side of advice or pointcut declarations, a regular Java identifier is allowed in certain pointcut designators in place of a type or collection of types. There are four primitive pointcut designators where this is allowed: `this`, `target`, and `args`. In all such cases, using an identifier rather than a type is as if the type selected was the type of the formal parameter, so that the pointcut

```
pointcut intArg(int i): args(i);
```

picks out join points where an `int` is being passed as an argument, but furthermore allows advice access to that argument.

Values can be exposed from named pointcuts as well, so

```
pointcut publicCall(int x): call(public *.*(int)) && intArg(x);
pointcut intArg(int i): args(i);
```

is a legal way to pick out all calls to public methods accepting an int argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type Object will also match primitive typed arguments, and expose a "boxed" version of the primitive. So,

```
pointcut publicCall(): call(public *.*(..)) && args(Object);
```

will pick out all unary methods that take, as their only argument, subtypes of Object (i.e., not primitive types like `int`), but

```
pointcut publicCall(Object o): call(public *.*(..)) && args(o);
```

will pick out all unary methods that take any argument: And if the argument was an `int`, then the value passed to advice will be of type `java.lang.Integer`.

## Primitive pointcuts

### Method−related pointcuts

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

`call(`*`Signature`*`)`

`execution(`*`Signature`*`)`
These two pointcuts also pick out constructor call end execution join points.

### Field−related pointcuts

AspectJ provides two primitive pointcut designators designed to capture field reference and assignment join points:

`get(`*`Signature`*`)`
`set(`*`Signature`*`)`
All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an `args` pointcut. So an aspect guarding an integer variable x declared in type T might be written as

```
aspect GuardedX {
    static final int MAX_CHANGE = 100;
    before(int newval): set(int T.x) && args(newval) {
        if (Math.abs(newval − T.x) > MAX_CHANGE)
            throw new RuntimeException();
    }
}
```

### Object creation−related pointcuts

AspectJ provides three primitive pointcut designators designed to capture the initializer execution join points of objects.

`call(`*`Signature`*`)`
`initialization(`*`Signature`*`)`
`execution(`*`Signature`*`)`
### Class initialization−related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

`staticinitialization(`*`TypePattern`*`)`
### Exception handler execution−related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

`handler(`*`TypePattern`*`)`
All handler join points are treated as having one argument, the value of the exception being handled, so at a handler join point, that value can be accessed with an `args` pointcut. So an aspect used to put FooException objects into some normal form before they are handled could be written as

```
aspect NormalizeFooException {
    before(FooException e): handler(FooException) && args(e) {
        e.normalize();
    }
}
```

### State−based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to discriminate, or pick out, join points. They may also be used to expose to advice the objects used for discrimination.

`this(`*`TypePattern`* `or` *`Id`*`)`
`target(`*`TypePattern`* `or` *`Id`*`)`
The this pointcut picks out all join points where the currently executing object (the object bound to `this`) is an instance of a particular type. The target pointcut picks out all join points where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type.

`args(`*`TypePattern`* `or` *`Id`* `or ".."`, `...)`
The args pointcut picks out all join points where the arguments are instances of some types. Each element in the comma−separated list is one of three things. If it is a type pattern, then the argument in that position must be an instance of a type of the type name. If it is an identifier, then the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to Object). If it is the special wildcard "..", then any number of arguments will match, just like in signatures. So the pointcut

```
args(int, .., String)
```

will pick out all join points where the first argument is an `int` and the last is a `String`.

### Control flow−based pointcuts

Some concerns cut across the control flow of the program. The cflow and cflowbelow primitive pointcut designators capture join points based on control flow.

`cflow(`*`Pointcut`*`)`
`cflowbelow(`*`Pointcut`*`)`
The cflow pointcut picks out all join points that occur between the start and the end of each of the pointcut's join points.

The cflowbelow pointcut picks out all join points that occur between the start and the end of each of the pointcut's join points, but not including the initial join point of the control flow itself.

### Program text−based pointcuts

While many concerns cut across the runtime structure of the program, some must deal with the actual lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

`within(`*`TypePattern`*`)`
`withincode(`*`Signature`*`)`
The within pointcut picks out all join points where the code executing is defined in the declaration of one of the types in *`TypePattern`*. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code within any of the type's inner types.

The withincode pointcut picks out all join points where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements and expressions of the method or constructor. It also includes any join points that are associated with code within any of the method or constructor's local or anonymous types.

### Dynamic property−based pointcuts

`if(`*`BooleanExpression`*`)`
The if pointcut picks out join points based on a dynamic property. It's syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out all call join points would be to use the pointcut

```
if(thisJoinPoint.getKind().equals("call"))
```

## Signatures

One very important property of a join point is its signature, which is used by many of AspectJ's pointcut designators to select particular join points.

At a method call join point, the signature is composed of the type used to access the method, the name of the method, and the the types of the called method's formal parameters and return value (if any).

At a method execution join point, the signature is composed of the type defining the method, the name of the method, and the the types of the executing method's formal parameters and return value (if any).

At a constructor call join point, the signature is composed of the type of the object to be constructed and the types of the called constructor's formal parameters.

At a constructor execution join point, the signature is composed of the type defining the constructor and the types of the executing constructor's formal parameters.

At an object initialization join point, the signature is composed of the type being initialized and the types of the formal parameters of the first constructor entered during the initialization of this type.

At an object pre−initialization join point, the signature is composed of the type being initialized and the types of the formal parameters of the first constructor entered during the initialization of this type.

At a field reference or assignment join point, the signature is composed of the type used to access or assign to the field, the name of the field, and the type of the field.

At a handler execution join point, the signature is composed of the exception type that the handler handles.

The `withincode`, `call`, `execution`, `get`, and `set` primitive pointcut designators all use signature patterns to determine the join points they describe. A signature pattern is an abstract description of one or more join−point signatures. Signature patterns are intended to match very closely the same kind of things one would write when defining individual methods and constructors.

Method definitions in Java include method names, method parameters, return types, modifiers like static or private, and throws clauses, while constructor definitions omit the return type and replace the method name

with the class name. The start of a particular method definition, in class `Test`, for example, might be

```
class C {
    public final void foo() throws ArrayOutOfBoundsException { ... }
}
```

In AspectJ, method signature patterns have all these, but most elements can be replaced by wildcards. So

```
call(public final void C.foo() throws ArrayOutOfBoundsException)
```

picks out call join points to that method, and the pointcut

```
call(public final void *.*() throws ArrayOutOfBoundsException)
```

picks out all call join points to methods, regardless of their name name or which class they are defined on, so long as they take no arguments, return no value, are both `public` and `final`, and are declared to throw `ArrayOutOfBounds` exceptions.

The defining type name, if not present, defaults to *, so another way of writing that pointcut would be

```
call(public final void *() throws ArrayOutOfBoundsException)
```

Formal parameter lists can use the wildcard `..` to indicate zero or more arguments, so

```
execution(void m(..))
```

picks out execution join points for void methods named `m`, of any number of arguments, while

```
execution(void m(.., int))
```

picks out execution join points for void methods named `m` whose last parameter is of type `int`.

The modifiers also form part of the signature pattern. If an AspectJ signature pattern should match methods without a particular modifier, such as all non−public methods, the appropriate modifier should be negated with the `!` operator. So,

```
withincode(!public void foo())
```

picks out all join points associated with code in null non−public void methods named `foo`, while

```
withincode(void foo())
```

picks out all join points associated with code in null void methods named `foo`, regardless of access modifier.

Method names may contain the * wildcard, indicating any number of characters in the method name. So

```
call(int *())
```

picks out all call join points to `int` methods regardless of name, but

```
call(int get*())
```

picks out all call join points to `int` methods where the method name starts with the characters "get".

AspectJ uses the `new` keyword for constructor signature patterns rather than using a particular class name. So the execution join points of private null constructor of a class C defined to throw an ArithmeticException can be picked out with

```
execution(private C.new() throws ArithmeticException)
```

## Type patterns

Type patterns are a way to pick out collections of types and use them in places where you would otherwise use only one type. The rules for using type patterns are simple.

### Type name patterns

First, all type names are also type patterns. So `Object`, `java.util.HashMap`, `Map.Entry`, `int` are all type patterns.

There is a special type name, *, which is also a type pattern. * picks out all types, including primitive types. So

```
call(void foo(*))
```

picks out all call join points to void methods named foo, taking one argument of any type.

Type names that contain the two wildcards "*" and "`..`" are also type patterns. The * wildcard matches zero or more characters characters except for ".", so it can be used when types have a certain naming convention. So

```
handler(java.util.*Map)
```

picks out the types java.util.Map and java.util.java.util.HashMap, among others, and

```
handler(java.util.*)
```

picks out all types that start with "`java.util.`" and don't have any more "."s, that is, the types in the `java.util` package, but not inner types (such as java.util.Map.Entry).

The "`..`" wildcard matches any sequence of characters that start and end with a ".", so it can be used to pick out all types in any subpackage, or all inner types. So

```
target(com.xerox..*)
```

picks out all join points where the target object is an instance of defined in any type beginning with "`com.xerox.`".

### Subtype patterns

It is possible to pick out all subtypes of a type (or a collection of types) with the "+" wildcard. The "+" wildcard follows immediately a type name pattern. So, while

```
call(Foo.new())
```

picks out all constructor call join points where an instance of exactly type Foo is constructed,

```
call(Foo+.new())
```

picks out all constructor call join points where an instance of any subtype of Foo (including Foo itself) is constructed, and the unlikely

```
call(*Handler+.new())
```

picks out all constructor call join points where an instance of any subtype of any type whose name ends in "Handler" is constructed.

### Array type patterns

A type name pattern or subtype pattern can be followed by one or more sets of square brackets to make array type patterns. So `Object[]` is an array type pattern, and so is `com.xerox..*[][]`, and so is `Object+[]`.

### Type patterns

Type patterns are built up out of type name patterns, subtype patterns, and array type patterns, and constructed with boolean operators `&&`, `||`, and `!`. So

```
staticinitialization(Foo || Bar)
```

picks out the static initializer execution join points of either Foo or Bar, and

```
call((Foo+ && ! Foo).new(..))
```

picks out the constructor call join points when a subtype of Foo, but not Foo itself, is constructed.

## Advice

```
before(Formals): Pointcut { Body }
after(Formals) returning [ (Formal) ]: Pointcut { Body }
after(Formals) throwing [ (Formal) ]: Pointcut { Body }
after(Formals) : Pointcut { Body }
Type around(Formals) [ throws TypeList ] : Pointcut { Body }
```

Advice defines crosscutting behavior. It is defined in terms of pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut. Exactly how the code runs depends on the kind of advice.

AspectJ supports three kinds of advice. The kind of advice determines how it interacts with the join points it is defined over. Thus AspectJ divides advice into that which runs before its join points, that which runs after its join points, and that which runs in place of (or "around") its join points.

While before advice is relatively unproblematic, there can be three interpretations of after advice: After the execution of a join point completes normally, after it throws an exception, or after it does either one. AspectJ allows after advice for any of these situations.

```
aspect A {
    pointcut publicCall(): call(public Object *(..));
    after() returning (Object o): publicCall() {
        System.out.println("Returned normally with " + o);
    }
    after() throwing (Exception e): publicCall() {
        System.out.println("Threw an exception: " + e);
    }
    after(): publicCall(){
        System.out.println("Returned or threw an Exception");
    }
}
```

After returning advice may not care about its returned object, in which case it may be written

```
after() returning: call(public Object *(..)) {
    System.out.println("Returned normally");
}
```

It is an error to try to put after returning advice on a join point that does not return the correct type. For example,

```
after() returning (byte b): call(int String.length()) {
    // this is an error
}
```

is not allowed. But if no return value is exposed, or the exposed return value is typed to `Object`, then it may be applied to any join point. If the exposed value is typed to `Object`, then the actual return value is converted to an object type for the body of the advice: `int` values are represented as `java.lang.Integer` objects, etc, and no value (from void methods, for example) is represented as `null`.

Around advice runs in place of the join point it operates over, rather than before or after it. Because around is allowed to return a value, it must be declared with a return type, like a method. A piece of around advice may be declared `void`, in which case it is not allowed to return a value, and instead whatever value the join point returned will be returned by the around advice (unless the around advice throws an exception of its own).

Thus, a simple use of around advice is to make a particular method constant:

```
aspect A {
    int around(): call(int C.foo()) {
        return 3;
    }
}
```

Within the body of around advice, though, the computation of the original join point can be executed with the special syntax

```
proceed( ... )
```

The proceed form takes as arguments the context exposed by the around's pointcut, and returns whatever the around is declared to return. So the following around advice will double the second argument to `foo` whenever it is called, and then halve its result:

```
aspect A {
    int around(int i): call(int C.foo(Object, int)) && args(i) {
        int newi = proceed(i*2)
```

```
        return newi/2;
    }
}
```

If the return value of around advice is typed to `Object`, then the result of proceed is converted to an object representation, even if it is originally a primitive value. And when the advice returns an Object value, that value is converted back to whatever representation it was originally. So another way to write the doubling and halving advice is:

```
aspect A {
    Object around(int i): call(int C.foo(Object, int)) && args(i) {
        Integer newi = (Integer) proceed(i*2)
        return new Integer(newi.intValue() / 2);
    }
}
```

In all kinds of advice, the parameters of the advice behave exactly like method parameters. In particular, assigning to any parameter affects only the value of the parameter, not the value that it came from. This means that

```
aspect A {
    after() returning (int i): call(int C.foo()) {
        i = i * 2;
    }
}
```

will *not* double the returned value of the advice. Rather, it will double the local parameter. Changing the values of parameters or return values of join points can be done by using around advice.

## Advice modifiers

The `strictfp` modifier is the only modifier allowed on advice, and it has the effect of making all floating−point expressions within the advice be FP−strict.

## Advice and checked exceptions

An advice declaration must include a `throws` clause listing the checked exceptions the body may throw. This list of checked exceptions must be compatible with each target join point of the advice, or an error is signalled by the compiler.

For example, in the following declarations:

```
import java.io.FileNotFoundException;

class C {
    int i;

    int getI() { return i; }
}

aspect A {
    before(): get(int C.i) {
        throw new FileNotFoundException();
```

```
    }
    before() throws FileNotFoundException: get(int C.i) {
        throw new FileNotFoundException();
    }
}
```

both pieces of advice are illegal. The first because the body throws an undeclared checked exception, and the second because field get join points cannot throw `FileNotFoundException`s.

The exceptions that each kind of join point in AspectJ may throw are:

*method call and execution*
: the checked exceptions declared by the target method's `throws` clause.

*constructor call and execution*
: the checked exceptions declared by the target constructor's `throws` clause.

*field get and set*
: no checked exceptions can be thrown from these join points.

*exception handler execution*
: the exceptions that can be thrown by the target exception handler.

*static initializer execution*
: no checked exceptions can be thrown from these join points.

*initializer execution, pre−initialization, and initialization*
: any exception that is in the throws clause of *all* constructors of the initialized class.

## Advice precedence

Multiple pieces of advice may apply to the same join point. In such cases, the resolution order of the advice is based on advice precedence.

### Determining precedence

There are a number of rules that determine whether a particular piece of advice has precedence over another when they advise the same join point.

If the two pieces of advice are defined in different aspects, then there are three cases:

If aspect A is declared such that it `dominates` aspect B, then all advice defined in A has precedence over all advice defined in B.

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 a `dominates` keyword, advice in a subaspect dominates advice in a superaspect.

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

If the two pieces of advice are defined in the same aspect, then there are two cases:

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

Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

These rules can lead to circularity, such as

```
aspect A {
    before(): execution(void main(String[] args)) {}
    after():  execution(void main(String[] args)) {}
    before(): execution(void main(String[] args)) {}
}
```

such circularities will result in errors signalled by the compiler.

### Effects of precedence

At a particular join point, advice is ordered by precedence.

A piece of `around` advice controls whether advice of lower precedence will run by calling `proceed`. The call to `proceed` will run the advice with next precedence, or the computation under the join point if there is no further advice.

A piece of `before` advice can prevent advice of lower precedence from running by throwing an exception. If it returns normally, however, then the advice of the next precedence, or the computation under the join pint if there is no further advice, will run.

Running `after returning` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation returned normally, the body of the advice will run.

Running `after throwing` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation threw an exception of an appropriate type, the body of the advice will run.

Running `after` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then the body of the advice will run.

## Reflective access to the join point

Three special variables are visible within bodies of advice: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`. Each is bound to an object that encapsulates some of the context of the advice's current or enclosing join point. These variables exist because some pointcuts may pick out very large collections of join points. For example, the pointcut

```
pointcut publicCall(): call(public * *(..));
```

picks out calls to many methods. Yet the body of advice over this pointcut may wish to have access to the method name or parameters of a particular join point.

`thisJoinPoint` is bound to a complete join point object, while `thisJoinPointStaticPart` is bound to a part of the join point object that includes less information, but for which no memory allocation is required on each execution of the advice.

`thisEnclosingJoinPointStaticPart` is bound to the static part of the join point enclosing the current join point. Only the static part of this enclosing join point is available through this mechanism.

Like standard Java reflection, which uses objects from the `java.lang.reflect` hierarchy, join point objects have types in a type hierarchy. The type of objects bound to `thisJoinPoint` is `org.aspectj.lang.JoinPoint`, while `thisStaticJoinPoint` is bound to objects of interface type `org.aspectj.lang.JoinPoint.StaticPart`.

# Static crosscutting

Advice declarations change the behavior of classes they crosscut, but do not change their static type structure. For crosscutting concerns that do operate over the static structure of type hierarchies, AspectJ provides forms of introduction.

Each introduction form is a member of the aspect defining it, but defines a new member of another type.

## Member introduction

A method introduction looks like

*Modifiers Type TypePattern* . *Id*(*Formals*) { *Body* }
abstract *Modifiers Type TypePattern* . *Id*(*Formals*);
The effect of such an introduction is to make all the types in TypePattern support the new method. Interfaces in TypePattern will support the new method as well, even if the method is neither public nor abstract, so the following is legal AspectJ code:

```
interface Iface {}

aspect A {
    private void Iface.m() {
        System.err.println("I'm a private method on an interface");
    }
    void worksOnI(Iface iface) {
        // calling a private method on an interface
        iface.m();
    }
}
```

A constructor introduction looks like

*Modifiers TypePattern*.new(*Formals*) { *Body* }
The effect of such an introduction is to make all the types in TypePattern support the new constructor. You cannot introduce a constructor onto an interface, so if TypePattern includes an interface type it is an error.

A field introduction looks like one of

*Modifiers Type TypePattern.Id = Expression*;
*Modifiers Type TypePattern.Id*;
The effect of such an introduction is to make all the types in TypePattern support the new field. Interfaces in TypePattern will support the new field as well, even if the field is neither public, nor static, nor final.

Any occurrence of the identifier `this` in the body of the constructor or method introduction, or in the initializer of a field introduction, refers to the target type from the *`TypePattern`* rather than to the aspect type.

## Access modifiers

Members may be introduced with access modifiers public or private, or the default package−protected (protected introduction is not supported).

The access modifier applies in relation to the aspect, not in relation to the target type. So a member that is privately introduced is visible only from code that is defined within the aspect introducing it. One that is package−protectedly introduced is visible only from code that is defined within the introducing aspect's package.

Note that privately introducing a method (which AspectJ supports) is very different from introducing a private method (which AspectJ previously supported). AspectJ does not allow the introduction of the private method "void writeObject(ObjectOutputStream)" required to implement the interface java.io.Serializable.

## Conflicts

Introduction may cause conflicts among introduced members and between introduced members and defined members.

Assuming `otherPackage` is not the package defining the aspect `A`, the code

```
aspect A {
    private Registry otherPackage.*.r;
    public void otherPackage.*.register(Registry r) {
        r.register(this);
        this.r = r;
    }
}
```

adds a field "`r`" to every type in otherPackage. This field is only accessible from the code inside of aspect `A`. The aspect also adds a "`register`" method to every type in `otherPackage`. This method is accessible everywhere.

If any type in `otherPackage` already defines a private or package−protected field "`r`", there is no conflict: The aspect cannot see such a field, and no code in `otherPackage` can see the introduced "`r`".

If any type in `otherPackage` defines a public field "`r`", there is a conflict: The expression

```
this.r = r
```

is an error, since it is ambiguous whether the introduced "`r`" or the public "`r`" should be used.

If any type in `otherPackage` defines any method "`register(Registry)`" there is a conflict, since it would be ambiguous to any code that could see such a defined method which "`register(Registry)`" method was applicable.

Conflicts are resolved as much as possible as per Java's conflict resolution rules:

A subclass can inherit multiple *fields* from its superclasses, all with the same name and type. However, it is an error to have an ambiguous *reference* to a field.

A subclass can only inherit multiple *methods* with the same name and argument types from its superclasses if only zero or one of them is concrete (i.e., all but one is abstract, or all are abstract).

## Extension and Implementation

An aspect may introduce a superinterface or superclass onto a type, with the declarations

`declare parents: `*`TypePattern`*` extends `*`TypeList`*`;`

`declare parents: `*`TypePattern`*` implements `*`TypeList`*`;`

For example, if an aspect wished to make a particular class runnable, it might add an appropriate `void run()` method, but it should also change the type of the class to specify that it fulfills the `Runnable` interface. In order to implement the methods in the `Runnable` interface, the `run()` method must be publically introduced:

```
aspect A {
    declare parents: SomeClass implements Runnable;
    public void SomeClass.run() { ... }
}
```

## Interfaces with members

Through the use of introduction, interfaces may now carry (non−public−static−final) fields and (non−public−abstract) methods that classes can inherit. Conflicts may occur from ambiguously inheriting members from a superclass and multiple superinterfaces.

Because interfaces may carry non−static initializers, the order of super−interface instantiation is observable. We fix this order with the following three properties: A supertype is initialized before a subtype, that initialized code runs only once, and initializers for supertypes run in left−to−right order. Consider the following hierarchy where {`Object`, `C`, `D`, `E`} are classes, {`M`, `N`, `O`, `P`, `Q`} are interfaces.

```
  Object  M   O
       \ / \ /
        C   N   Q
         \ /   /
          D   P
           \ /
            E
```

when a new `E` is instantiated, the initializers run in this order:

```
  Object M C O N D Q P E
```

## Warnings and Errors

An aspect may specify that a particular join point should never be reached.

```
declare error: Pointcut: String;
declare warning: Pointcut: String;
```

If the compiler determines that a join point in *`Pointcut`* could possibly be reached, then it will signal either an error or warning, as declared, using the *`String`* for its message.

## Softened exceptions

An aspect may specify that a particular kind of exception, if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as a `org.aspectj.lang.SoftException`, which is subtype of `RuntimeException` and thus does not need to be declared.

```
declare soft: TypePattern: Pointcut;
```

For example, the aspect

```
aspect A {
    declare soft: Exception: execution(void main(String[] args));
}
```

Would, at the execution join point, catch any `Exception` and rethrow a `org.aspectj.lang.SoftException` containing original exception.

This is similar to what the following advice would do

```
aspect A {
    void around() execution(void main(String[] args)) {
        try { proceed(); }
        catch (Exception e) {
            throw new org.aspectj.lang.SoftException(e);
        }
    }
}
```

except, in addition to wrapping the exception, it also effects Java's static exception checking mechanism.

## Statically determinable pointcuts

Pointcuts that appear inside of `declare` forms have certain restrictions. Like other pointcuts, these pick out join points, but they do so in a way that is statically determinable.

Consequently, such pointcuts may not include, directly or indirectly (through user−defined pointcut declarations) pointcuts that discriminate based on dynamic (runtime) context. Therefore, such pointcuts may not be defined in terms of

cflow

cflowbelow

this
target
args
if
all of which can discriminate on runtime information.

# Aspects

An aspect is a crosscutting type defined by the aspect declaration. The aspect declaration is similar to the class declaration in that it defines a type and an implementation for that type. It differs in that the type and implementation can cut across other types (including those defined by other aspect declarations), and that it may not be directly instantiated with a new expression, with cloning, or with serialization. Aspects may have one constructor definition, but if so it must be of a constructor taking no arguments and throwing no checked exceptions.

Aspects may be defined either at the package level, or as an inner aspect, that is, a member of a class, interface, or aspect. If it is not at the package level, the aspect *must* be defined with the static keyword. Local and anonymous aspects are not allowed.

## Aspect Extension

To support abstraction and composition of crosscutting concerns, aspects can be extended in much the same way that classes can. Aspect extension adds some new rules, though.

### Aspects may extend classes and implement interfaces

An aspect, abstract or concrete, may extend a class and may implement a set of interfaces. Extending a class does not provide the ability to instantiate the aspect with a new expression: The aspect may still only define a null constructor.

### Classes may not extend aspects

It is an error for a class to extend or implement an aspect.

### Aspects extending aspects

Aspects may extend other aspects, in which case not only are fields and methods inherited but so are pointcuts. However, aspects may only extend abstract aspects. It is an error for a concrete aspect to extend another concrete aspect.

## Aspect instantiation

Unlike class expressions, aspects are not instantiated with `new` expressions. Rather, aspect instances are automatically created to cut across programs.

Because advice only runs in the context of an aspect instance, aspect instantiation indirectly controls when advice runs.

The criteria used to determine how an aspect is instantiated is inherited from its parent aspect. If the aspect has no parent aspect, then by default the aspect is a singleton aspect.

### Singleton Aspects

```
aspect Id { ... }
aspect Id issingleton { ... }
```

By default, or by using the modifier `issingleton`, an aspect has exactly one instance that cuts across the entire program. That instance is available at any time during program execution with the static method `aspectOf()` defined on the aspect −− so, in the above examples, `A.aspectOf()` will return A's instance. This aspect instance is created as the aspect's classfile is loaded.

Because the an instance of the aspect exists at all join points in the running of a program (once its class is loaded), its advice will have a chance to run at all such join points.

### Per−object aspects

```
aspect Id perthis(Pointcut) { ... }
aspect Id pertarget(Pointcut) { ... }
```

If an aspect A is defined `perthis(Pointcut)`, then one object of type A is created for every object that is the executing object (i.e., "this") at any of the join points picked out by `Pointcut`. The advice defined in A may then run at any join point where the currently executing object has been associated with an instance of A.

Similarly, if an aspect A is defined `pertarget(Pointcut)`, then one object of type A is created for every object that is the target object of the join points picked out by `Pointcut`. The advice defined in A may then run at any join point where the target object has been associated with an instance of A.

In either case, the static method call `A.aspectOf(Object)` can be used to get the aspect instance (of type A) registered with the object. Each aspect instance is created as early as possible, but not before reaching a join point picked out by `Pointcut` where there is no associated aspect of type A.

Both `perthis` and `pertarget` aspects may be affected by code the AspectJ compiler controls, as discussed in the Implementation Limitations appendix.

### Per−control−flow aspects

```
aspect Id percflow(Pointcut) { ... }
aspect Id percflowbelow(Pointcut) { ... }
```

If an aspect A is defined `percflow(`*`Pointcut`*`)` or `percflowbelow(`*`Pointcut`*`)`, then one object of type A is created for each flow of control of the join points picked out by *`Pointcut`*, either as the flow of control is entered, or below the flow of control, respectively. The advice defined in A may run at any join point in or under that control flow. During each such flow of control, the static method `A.aspectOf()` will return an object of type A. An instance of the aspect is created upon entry into each such control flow.

## Aspect privilege

`privileged aspect` *`Id`* `{ ... }`

Code written in aspects is subject to the same access control rules as Java code when referring to members of classes or aspects. So, for example, code written in an aspect may not refer to members with default (package−protected) visibility unless the aspect is defined in the same package.

While these restrictions are suitable for many aspects, there may be some aspects in which advice or introductions needs to access private or protected resources of other types. To allow this, aspects may be declared `privileged`. Code in priviliged aspects has access to all members, even private ones.

```
class C {
    private int i = 0;
    void incI(int x) { i = i+x; }
}
privileged aspect A {
    static final int MAX = 1000;
    before(int x, C c): call(void C.incI(int)) && target(c) && args(x) {
        if (c.i+x > MAX) throw new RuntimeException();
    }
}
```

In this case, if A had not been declared privileged, the field reference c.i would have resulted in an error signalled by the compiler.

If a privileged aspect can access multiple versions of a particular member, then those that it could see if it were not privileged take precedence. For example, in the code

```
class C {
    private int i = 0;
    void foo() { }
}
privileged aspect A {
    private int C.i = 999;
    before(C c): call(void C.foo()) target(c) {
        System.out.println(c.i);
    }
}
```

A's introduced private field C.i, initially bound to 999, will be referenced in the body of the advice in preference to C's privately declared field, since the A would have access to fields it introduces even if it were not privileged.

### Aspect domination

`aspect` *`Id`* `dominates` *`TypePattern`* `{ ... }`

An aspect may declare that the advice in it dominates the advice in some other aspect. Such declarations are like the `strictfp` keyword in Java; it applies to the advice declarations inside of the respective aspects, and states that the advice declared in the current aspect has more precedence than the advice in the aspects from *`TypePattern`*.

For example, the CountEntry aspect might want to count the entry to methods in the current package accepting a Type object as its first argument. However, it should count all entries, even those that the aspect DisallowNulls causes to throw exceptions. This can be accomplished by stating that CountEntry dominates DisallowNulls.

```
aspect DisallowNulls {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);
    before(Type obj):  allTypeMethods(obj) {
        if (obj == null) throw new RuntimeException();
    }
}
aspect CountEntry dominates DisallowNulls {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);
    static int count = 0;
    before():  allTypeMethods(Type) {
        count++;
    }
}
```

# Appendix C. Implementation Limitations

Certain elements of AspectJ's semantics are difficult to implement without making modifications to the virtual machine. One way to deal with this problem would be to specify only the behavior that is easiest to implement. We have chosen a somewhat different approach, which is to specify an ideal language semantics, as well as a clearly defined way in which implementations are allowed to deviate from that semantics. This makes it possible to develop conforming AspectJ implementations today, while still making it clear what later, and presumably better, implementations should do tomorrow.

According to the AspectJ language semantics, the declaration

```
before(): get(int Point.x) { System.out.println("got x"); }
```

should advise all accesses of a field of type int and name x from instances of type (or subtype of) Point. It should do this regardless of whether all the source code performing the access was available at the time the aspect containing this advice was compiled, whether changes were made later, etc.

But AspectJ implementations are permitted to deviate from this in a well−defined way −− they are permitted to advise only accesses in *code the implementation controls*. Each implementation is free within certain bounds to provide its own definition of what it means to control code.

In the current AspectJ compiler, ajc, control of the code means having source code for an aspect and all the code it should advise available at compile time. This means that if some class Client contains code with the expression `new Point().x` (which results in a field get join point at runtime), the current AspectJ compiler will fail to advise that access unless Client.java is compiled at the same the aspect is compiled. It

also means that join points associated with code in precompiled libraries (such as java.lang) can not be advised.

Different join points have different requirements. Method call join points can be advised only if ajc controls *either* the code for the caller or the code for the receiver. Constructor call join points can be advised only if ajc controls the code for the caller. Field reference or assignment join points can be advised only if ajc controls the code for the "caller", the code actually making the reference or assignment. Initialization join points can be advised only if ajc controls the code of the type being initialized, and execution join points can be advised only if ajc controls the code for the method or constructor body in question.

Aspects that are defined `perthis` or `pertarget` also have restrictions based on control of the code. In particular, at a join point where the source code for the currently executing object is not available, an aspect defined `perthis` of that join point will not be associated. So aspects defined `perthis(Object)` will not create aspect instances for every object, just those whose class the compiler controls. Similar restrictions apply to `pertarget` aspects.

Other AspectJ implementations, indeed, future versions of ajc, may define *code the implementation controls* more liberally.

Control may mean that classes need only be available in classfile or jarfile format. So, even if Client.java and Point.java were precompiled, their join points could still be advised. In such a system, though, it might still be the case that join points from code of system libraries such as java.lang could not be advised.

Or control could even include system libraries, thus allowing a call join point from java.util.Hashmap to java.lang.Object to be advised.

All AspectJ implementations are required to control the code of the files that the compiler compiles itself.

The important thing to remember is that core concepts of AspectJ, such as the join point, are unchanged, regardless of which implementation is used. During your development, you will have to be aware of the limitations of the ajc compiler you're using, but these limitations should not drive the design of your aspects.

# Appendix D. Glossary

*advice*
: Code, similar to a method, that is executed when a join point is reached. There are three kinds of advice: before advice that runs when a join point is reached, but before the method in question executes, after advice that executes after the method body executes, but before control returns to the caller, and around advice that runs before and after the method in question runs, and also has explicit control over whether the method is run at all.

*AOP*
: See *aspect−oriented programming*.

*aspect*
: A modular unit of crosscutting implementation in *aspect−oriented programming*, just as classes are the modular unit of implementation in object−oriented programming.

*aspect−oriented programming*

A type or style of programming that explicitly takes into account *crosscutting concerns*, just as object−oriented programming explicitly takes into account classes and objects.

*crosscutting concerns*
Issues or programmer concerns that are not local to the natural unit of modularity.

*dynamic context*
The state of a program while it is executing. Contrast with *lexical context*.

*join point*
A well−defined instant in the execution of a program. In AspectJ, join points are also principled, i.e. not every possible instance in the execution of a program is a potential join point.

*lexical context*
The state of a program as it is written. Contrast with *dynamic context*.

*name−based pointcut designator*
A type of pointcut designator that enumerates and composes explicitly named join points. For example,

```
pointcut move():
  call(void FigureElement.setXY(int,int)) ||
  call(void Point.setX(int))              ||
  call(void Point.setY(int))              ||
  call(void Line.setP1(Point))            ||
  call(void Line.setP2(Point));
```

is a pointcut designator that explicitly names five join points. See also *property−based pointcut designator*.

*pointcut*
A collection of join points.

*pointcut designator*
The name of a pointcut, or an expression which identifies a pointcut. Pointcut designators can be primitive or composite. Composite pointcut designators are primitive pointcut designators composed using the operators `||`, `&&`, `and` `!`. See also *name−based pointcut designator* and *property−based pointcut sesignator*.

*post−condition*
A test or assertion that must be true after a method has executed.

*pre−condition*
A test or assertion that must be true when a method is called.

*property−based pointcut designator*
A type of pointcut designator that specifies pointcuts in terms of the properties of methods rather than just their names. For example,

```
call(public * Figure.*(..))
```

specifies all the public methods in the class `Figure` regardless of the type and number of their

arguments or return type. See also *name−based pointcut designator*.

*reusable aspect*
: An aspect that can be extended or inherited from.

*signature*
: The number, order and type of the arguments to a method.

`thisJoinPoint`
: The special variable that identifies the current join point when a non−static join point is reached.

# Bibliography

Doug Lea. *Concurrent Programming in Java, Second Edition*. Addison−Wesley. Reading, MA. Copyright © 1999.

Gregor Kiczales, et al. *An Overview of AspectJ*. in Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP), Springer. Budapest, Hungary. Copyright © 2001.

Betrand Meyer. *Object−Oriented Software Construction, 2/e*. Prentice−Hall. New York, NY. Copyright © 1999.