Declarative, Formal, and Extensible Syntax Definition for AspectJ

A Case for Scannerless Generalized-LR Parsing

Martin Bravenboer Éric Tanter Eelco Visser

Software Engineering Research Group
Department of Software Technology
Delft University of Technology
The Netherlands

DCC, University of Chile
Chile

October 25, 2006

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem? 1

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

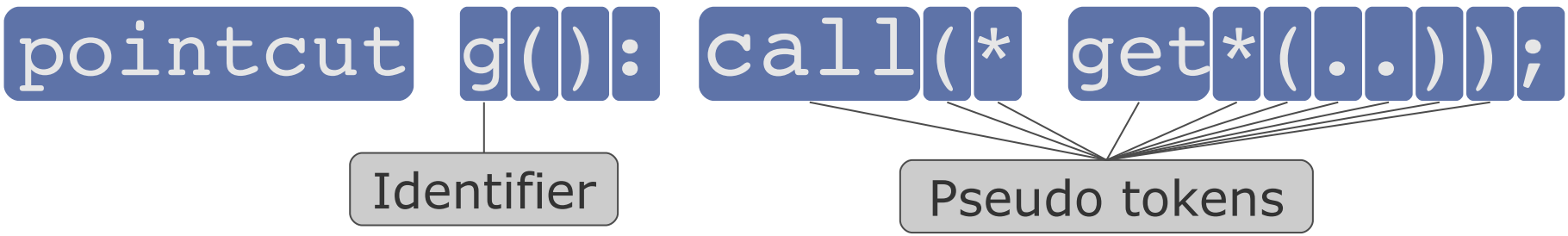
```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

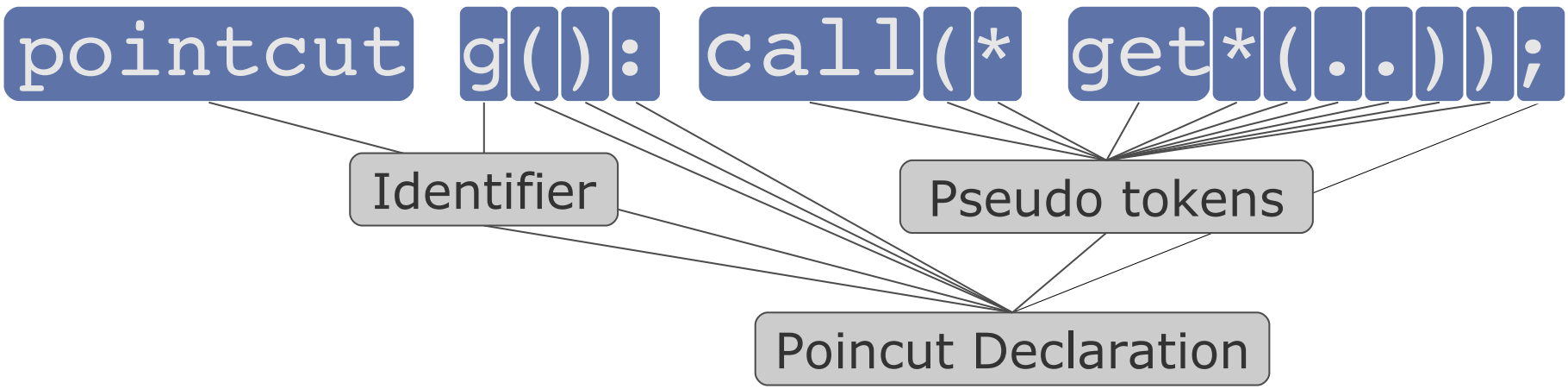
Keywords cannot be reserved globally

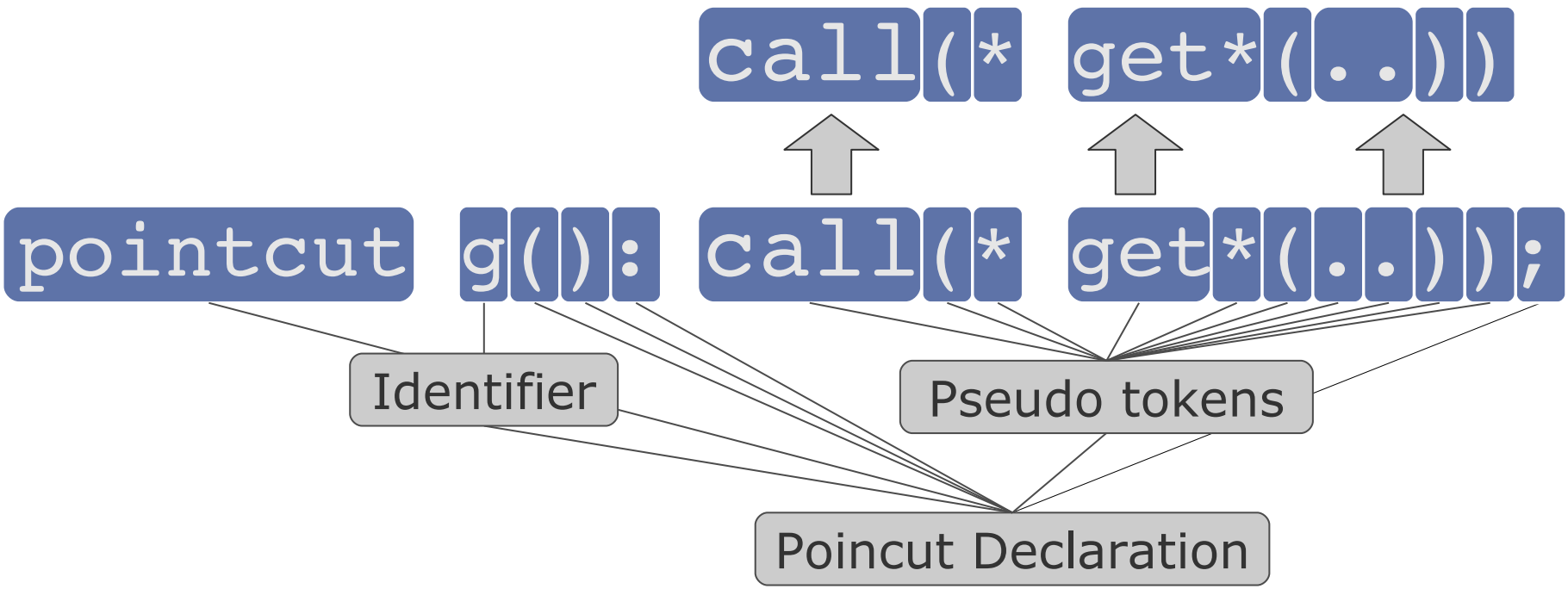
- `get(* Foo.*)`, `target(e)`, `args(event)` ...

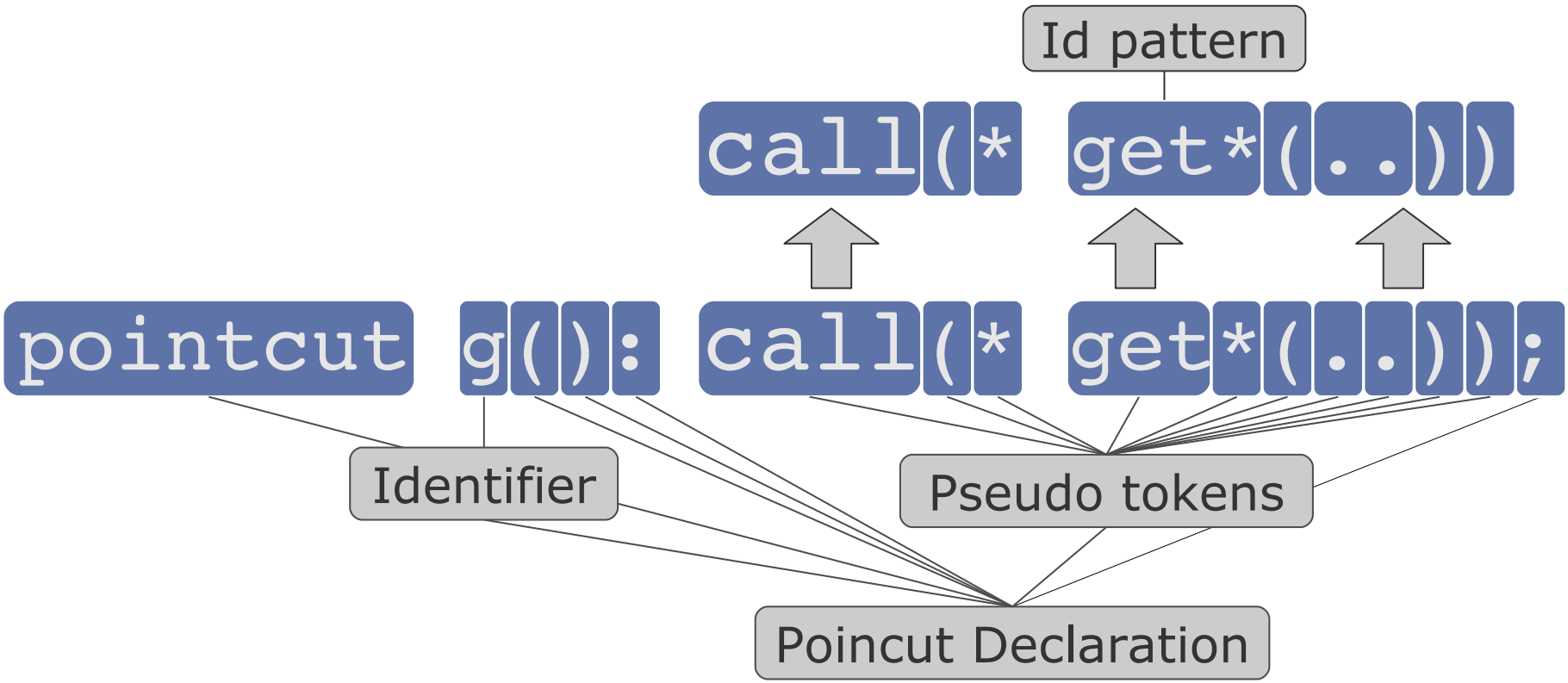
```
pointcut g(): call(* get*(..));
```

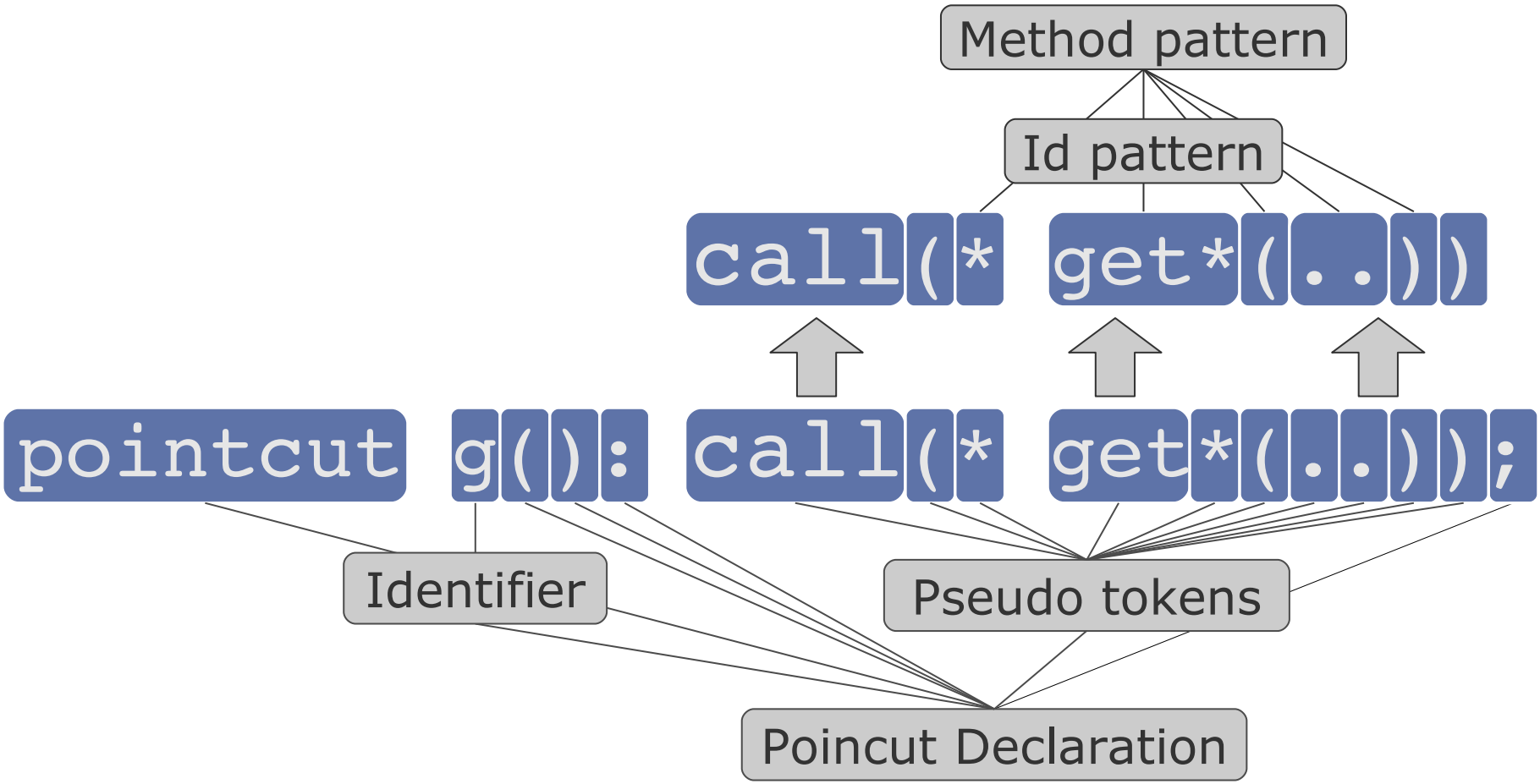
```
pointcut g(): call(* get*(..));
```

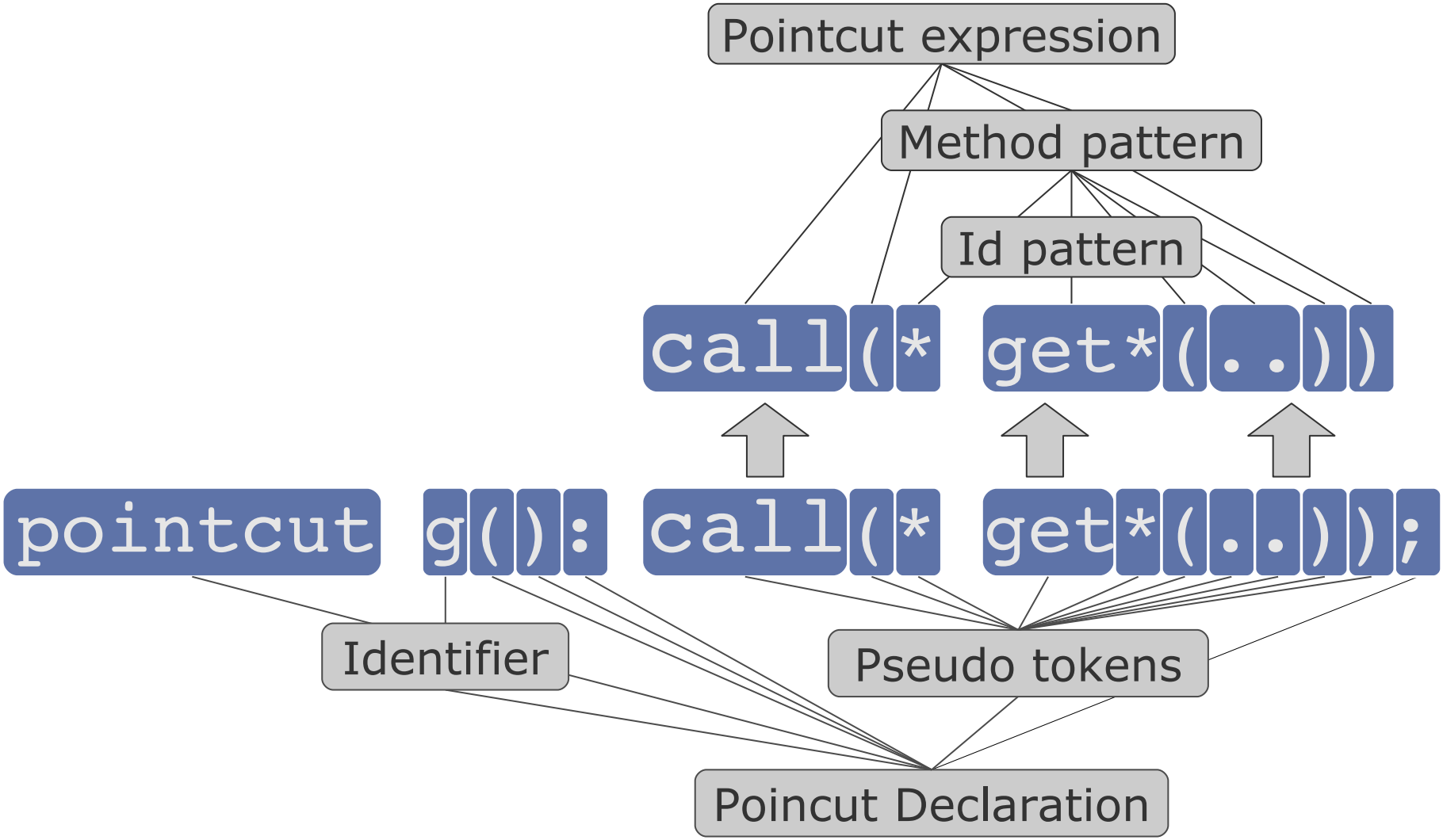

pointcut g(): call(* get*(..));
Identifier
Pseudo tokens

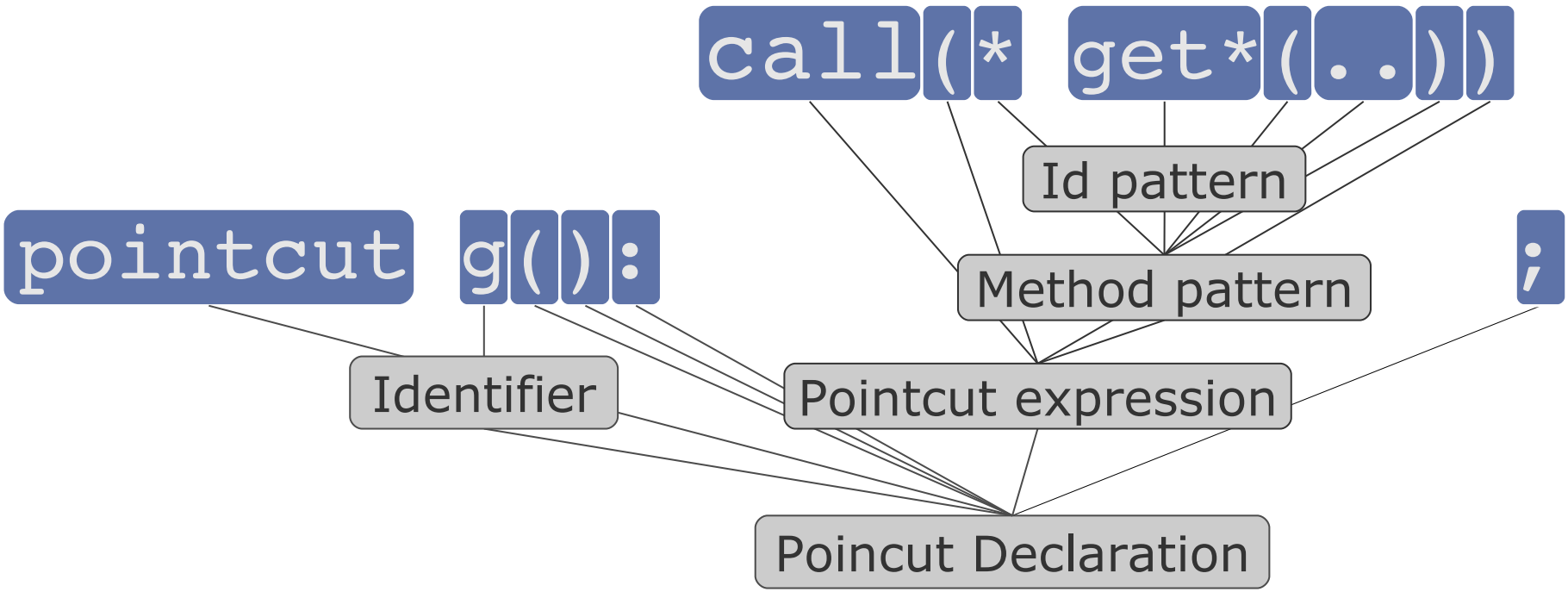

pointcut g(): call(* get*(..));
Identifier
Pseudo tokens
Poincut Declaration


call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration


Id pattern
call(* get*(..))
pointcut g(): call(* get*(..));
Identifier
Pseudo tokens
Poincut Declaration


Method pattern
Id pattern
call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration


Pointcut expression
Method pattern
Id pattern
call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration


call(* get*(..))
Id pattern
pointcut g():
Method pattern
;
Identifier
Pointcut expression
Poincut Declaration

- **More invalid tokenizations**

```
call(* *1.Function+.apply(..))
call(* *1.F unction+.apply(..))
```

- **Invalid error report**

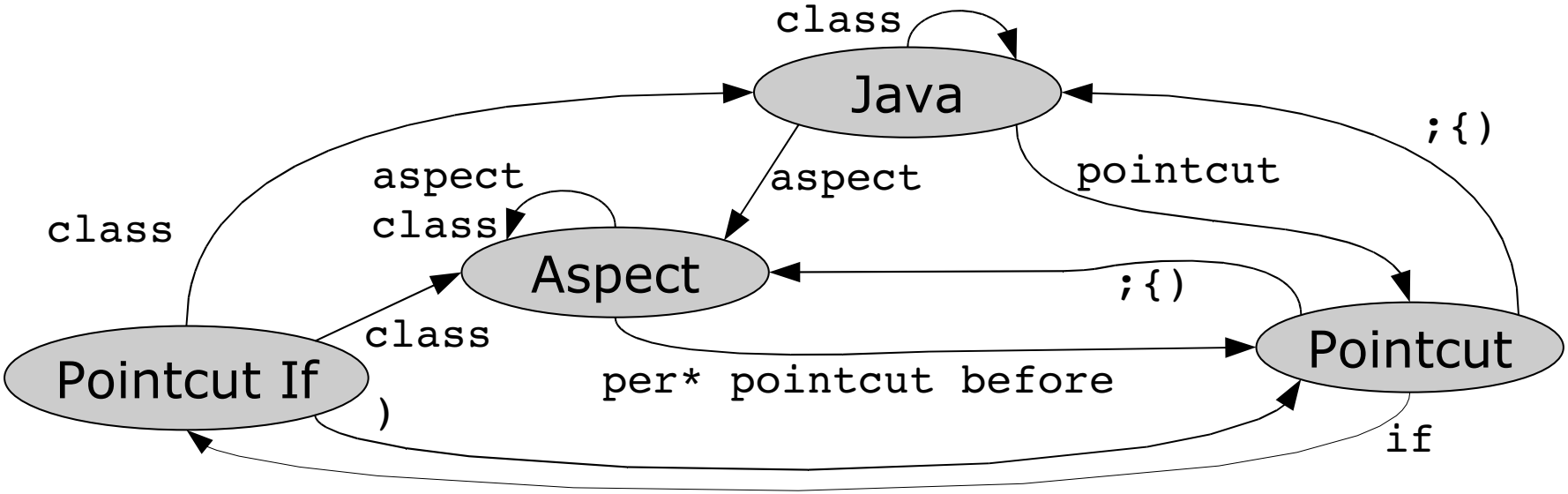
```
call(void *0.Ef())
ajc: [error] Invalid float literal number
```

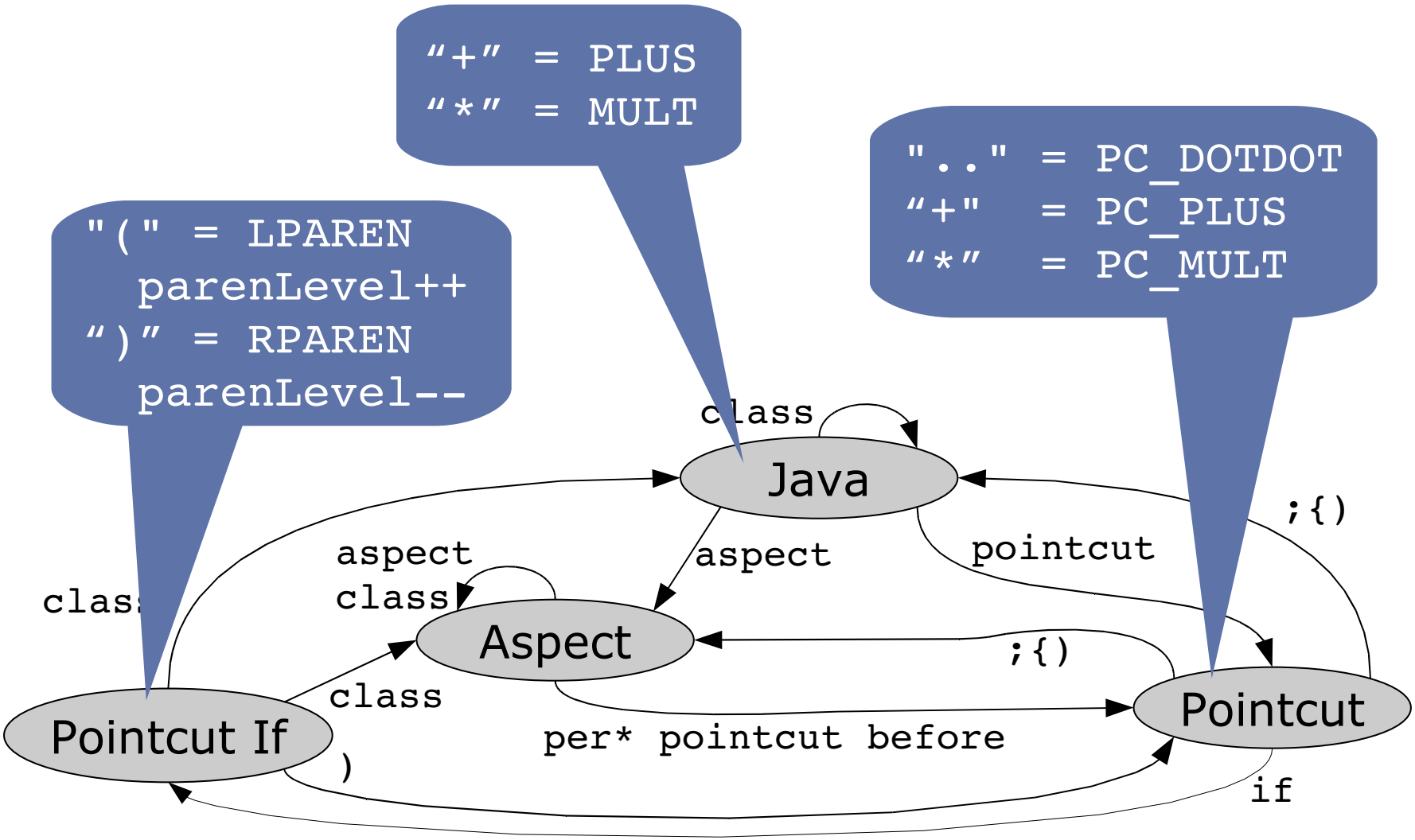
- **If not allowed in method pattern**

```
call(boolean *.for*(..)) || call(boolean *.if*(..))
ajc: [error] Syntax error on token "if"
```

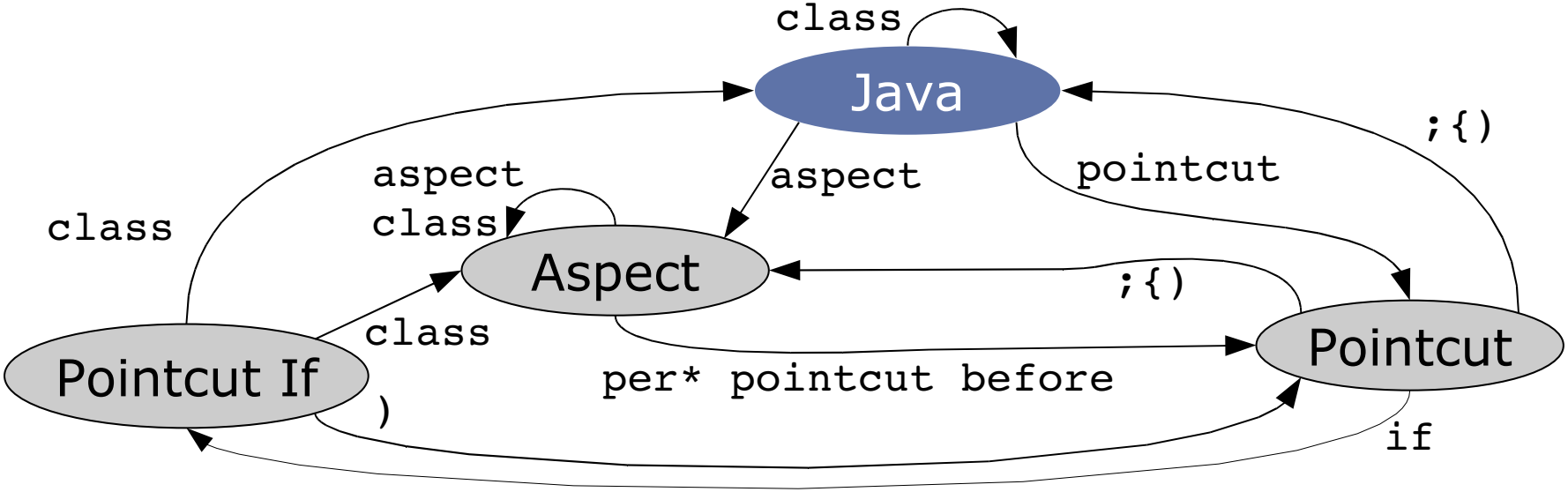
- **Tokenized in same way**

```
call(* *1.foo(..)) && call(* *1.f oo(..))
```

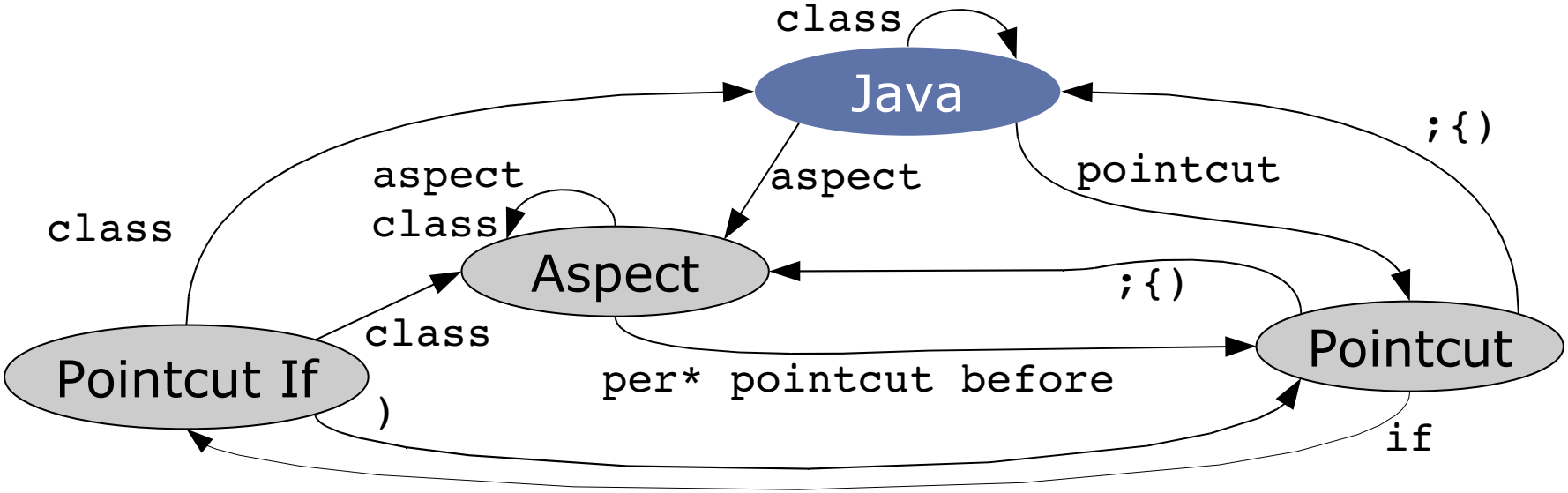

class
Java
;{)
aspect
class
aspect
pointcut
class
Aspect
;{)
class
Pointcut If
per* pointcut before
Pointcut
)
if


"+" = PLUS
"*" = MULT
".." = PC_DOTDOT
"+" = PC_PLUS
"*" = PC_MULT
"(" = LPAREN
parenLevel++
")" = RPAREN
parenLevel--
class
Java
aspect
class
aspect
pointcut
;{}
clas
Aspect
;{}
class
Pointcut If
per* pointcut before
Pointcut
)
if

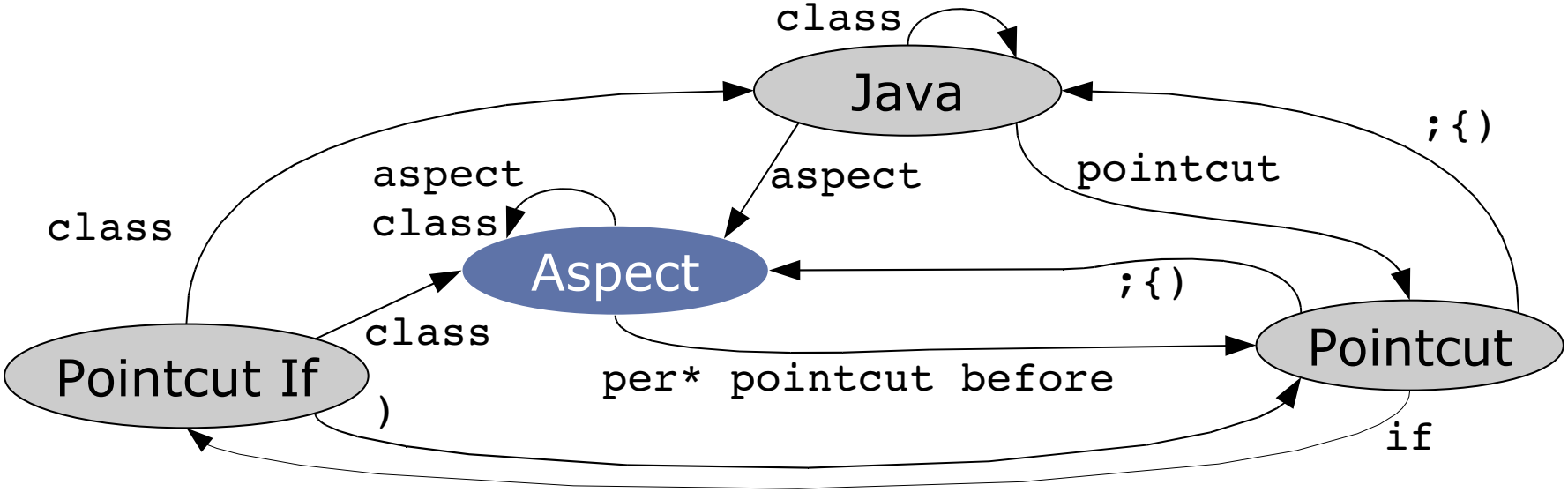
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



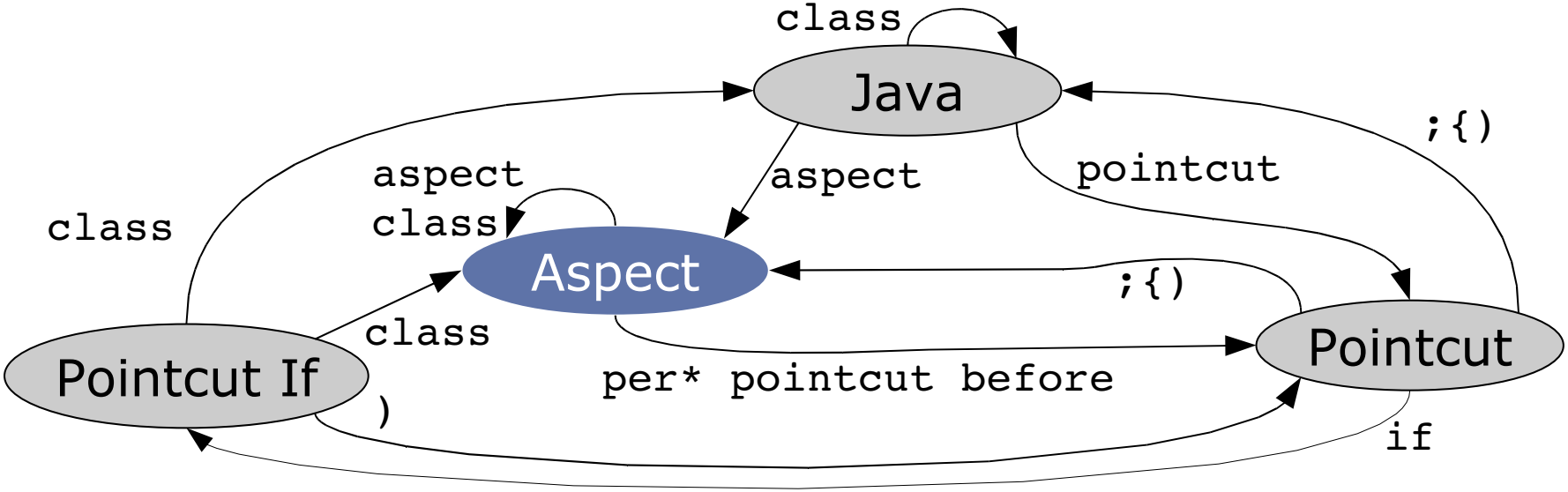
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



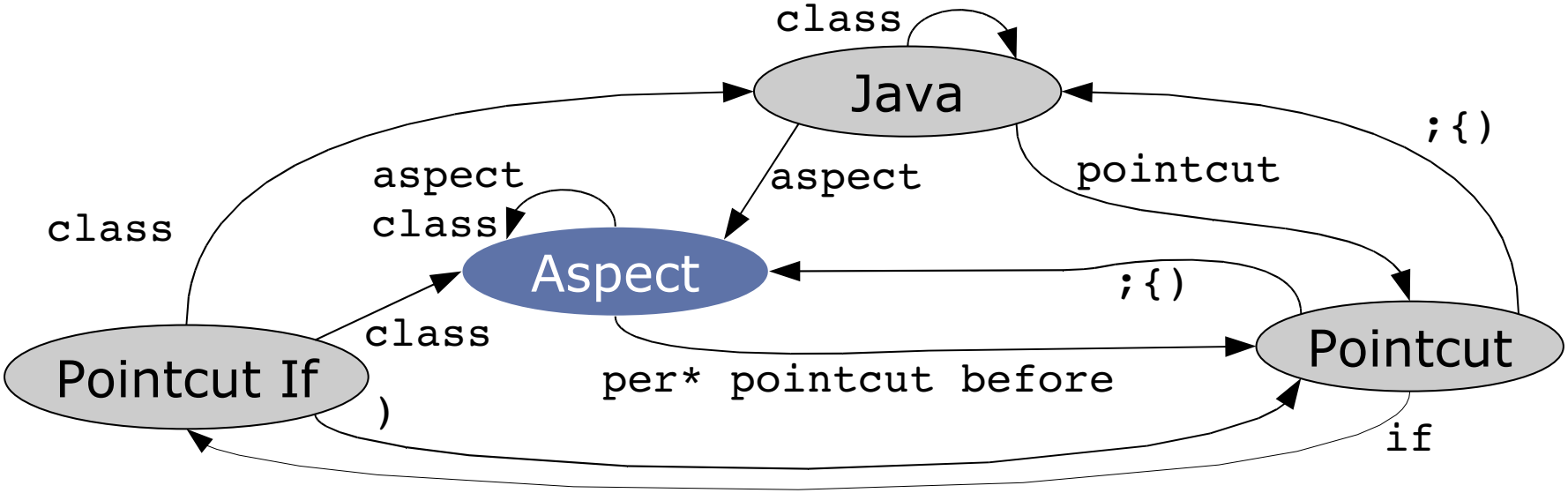
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



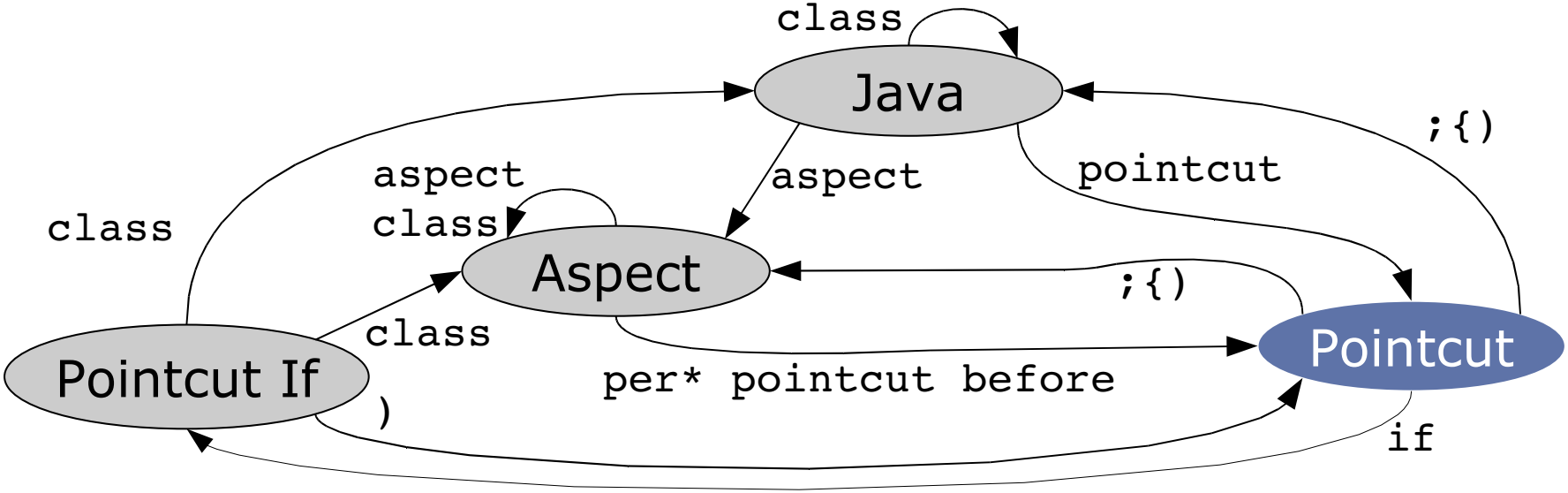
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



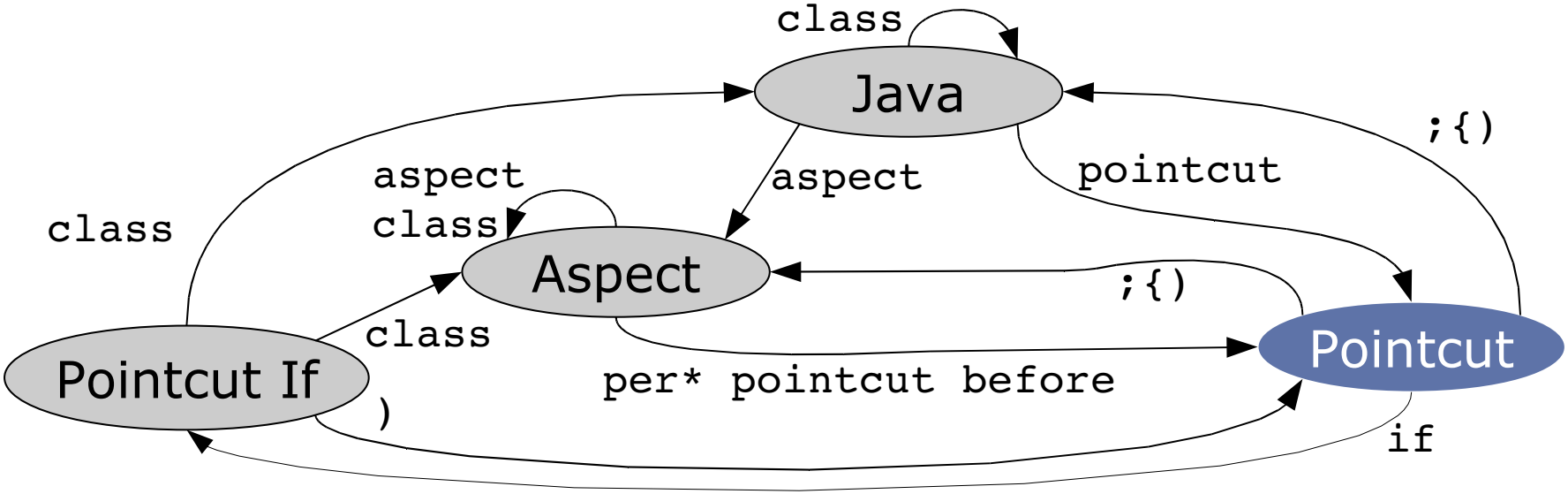
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



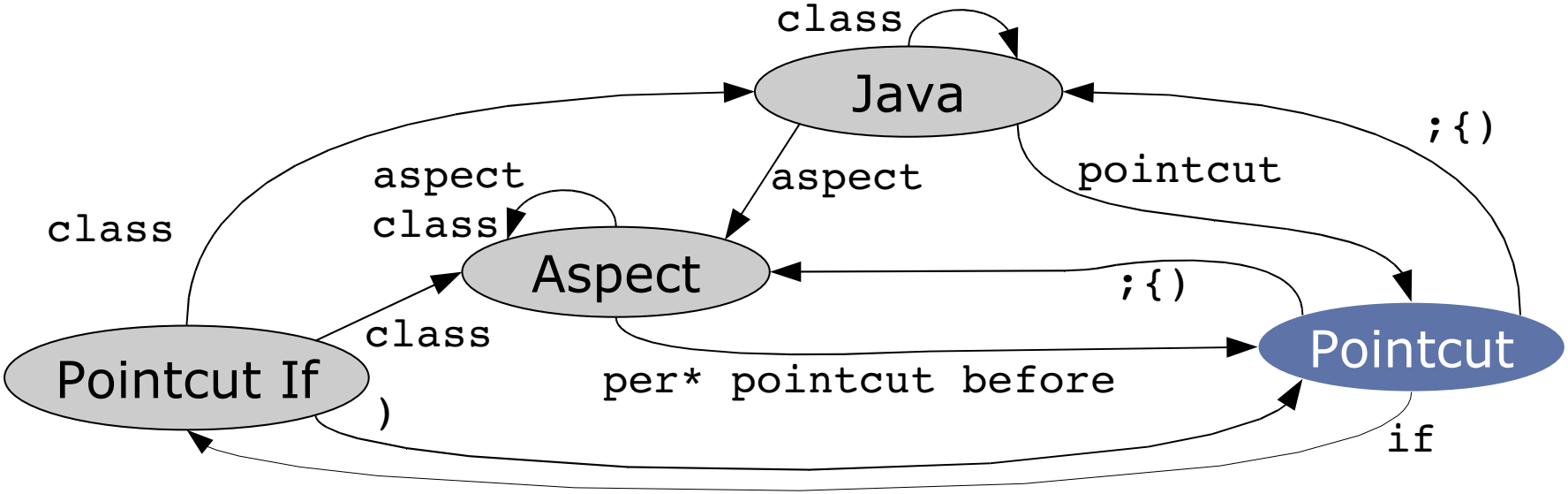
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



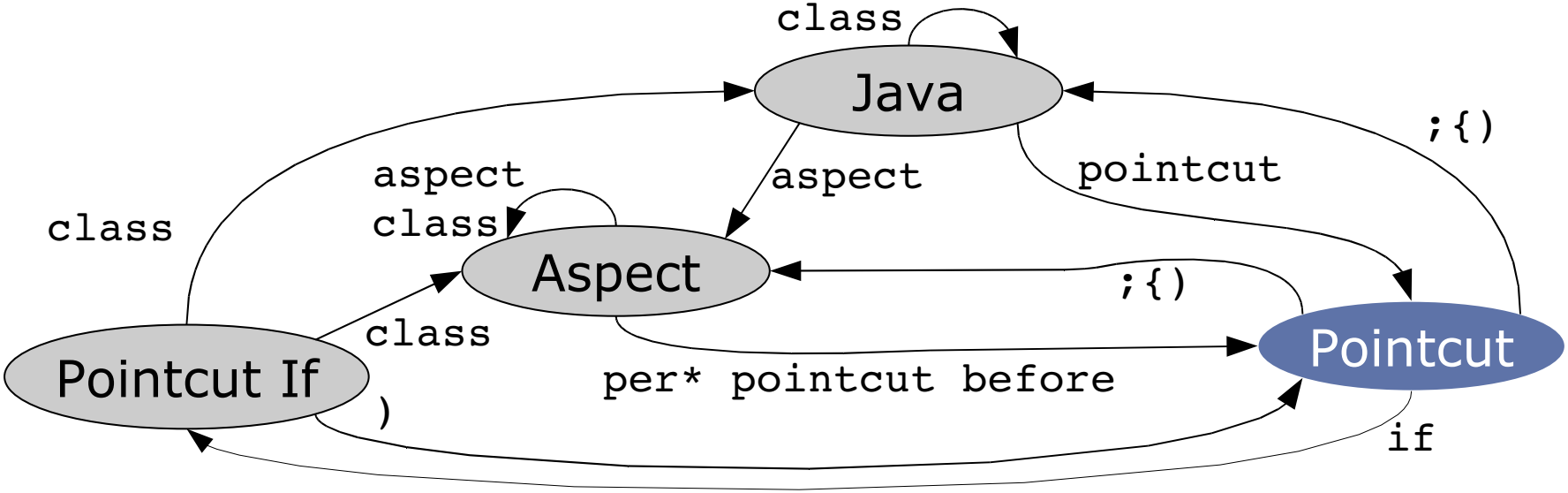
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



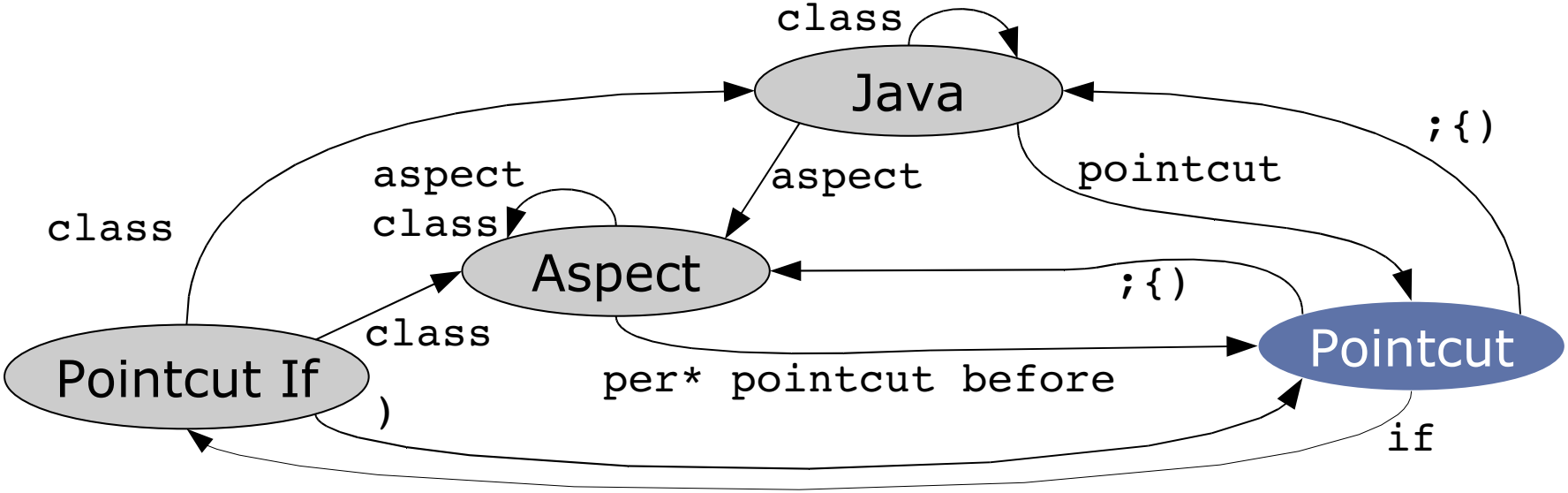
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



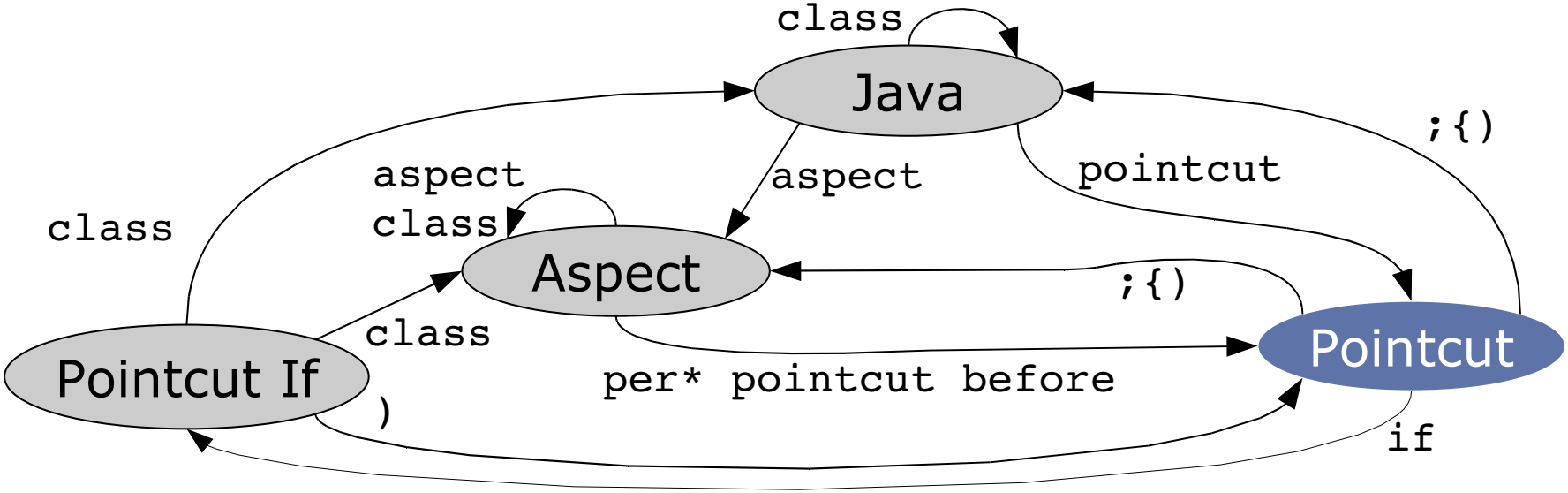
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



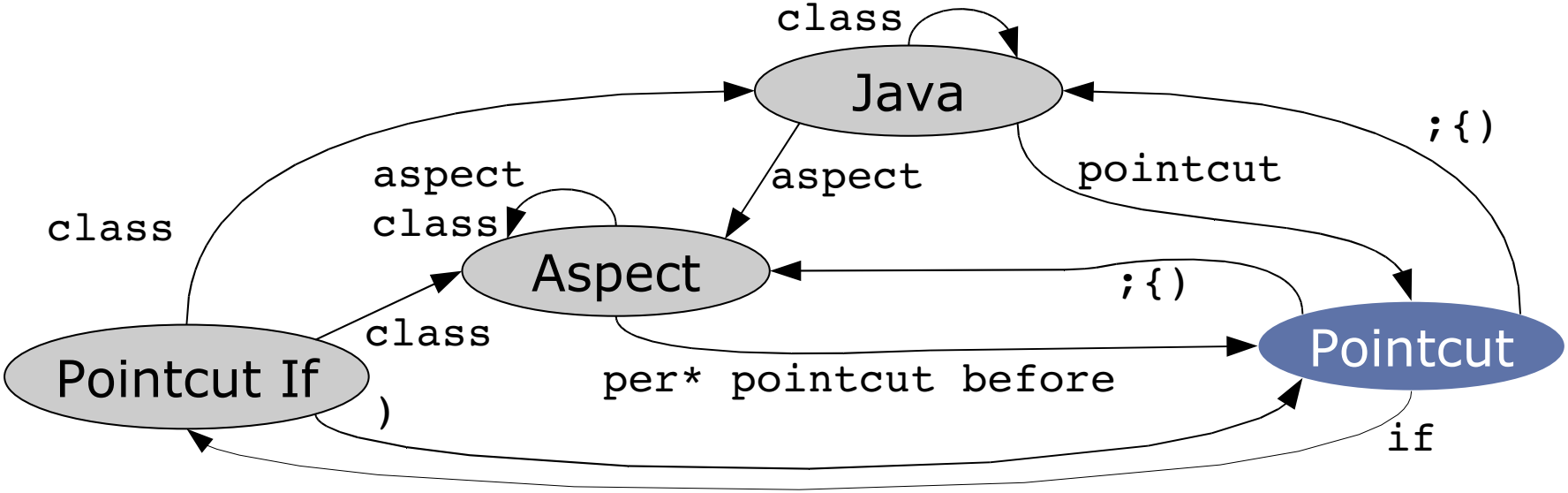
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



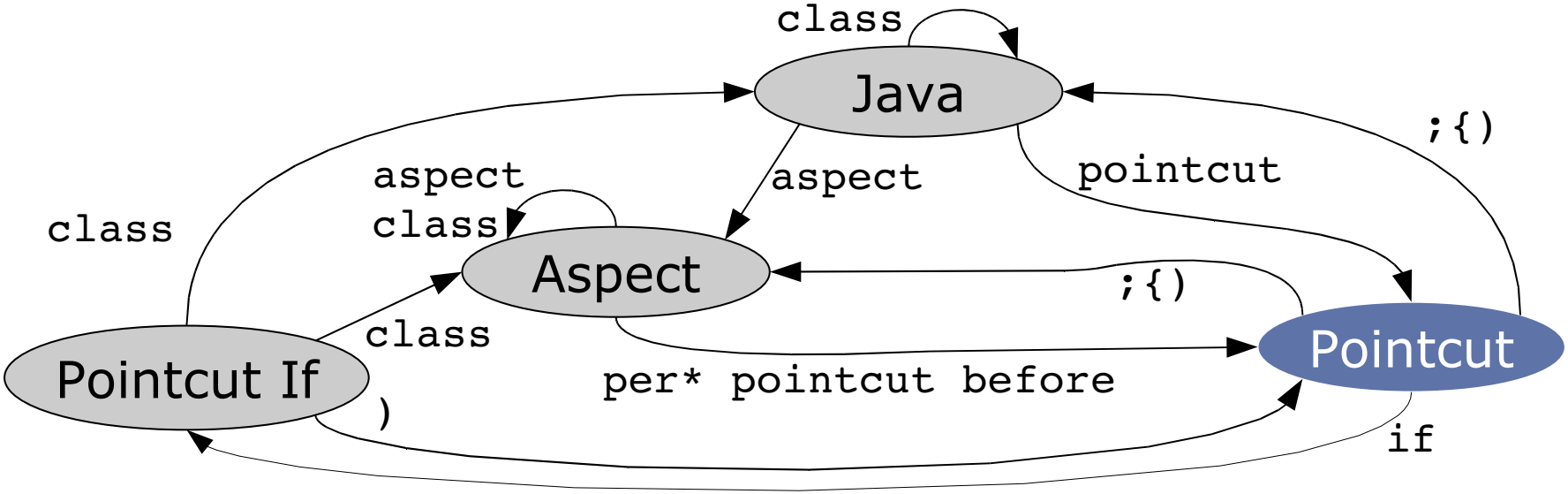
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



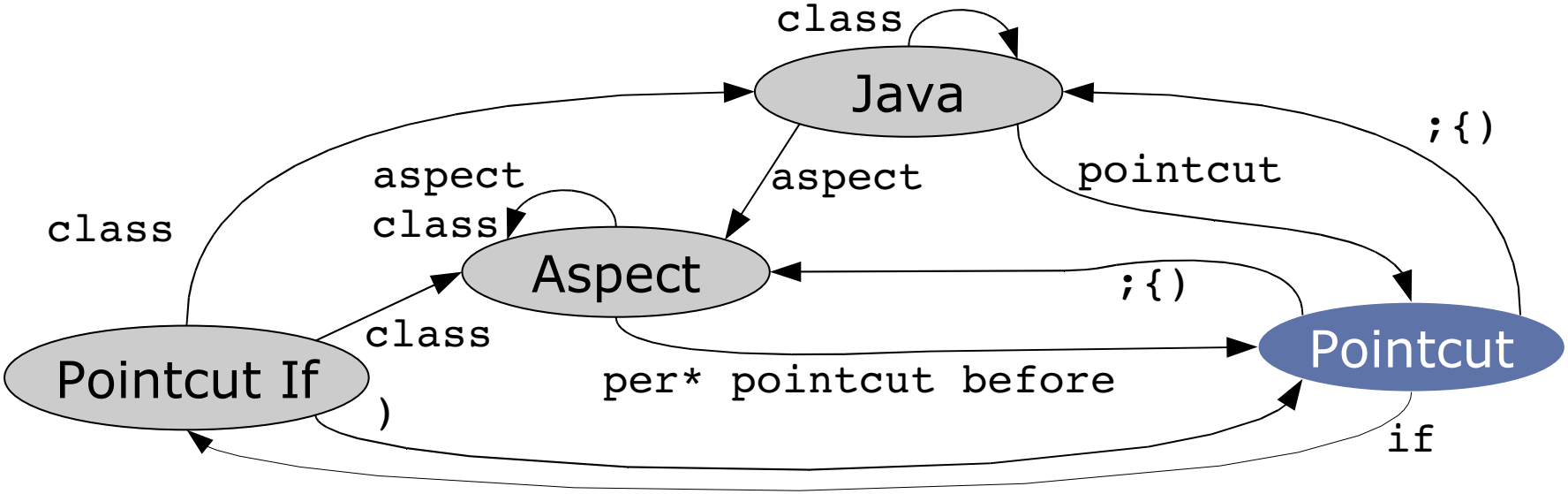
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



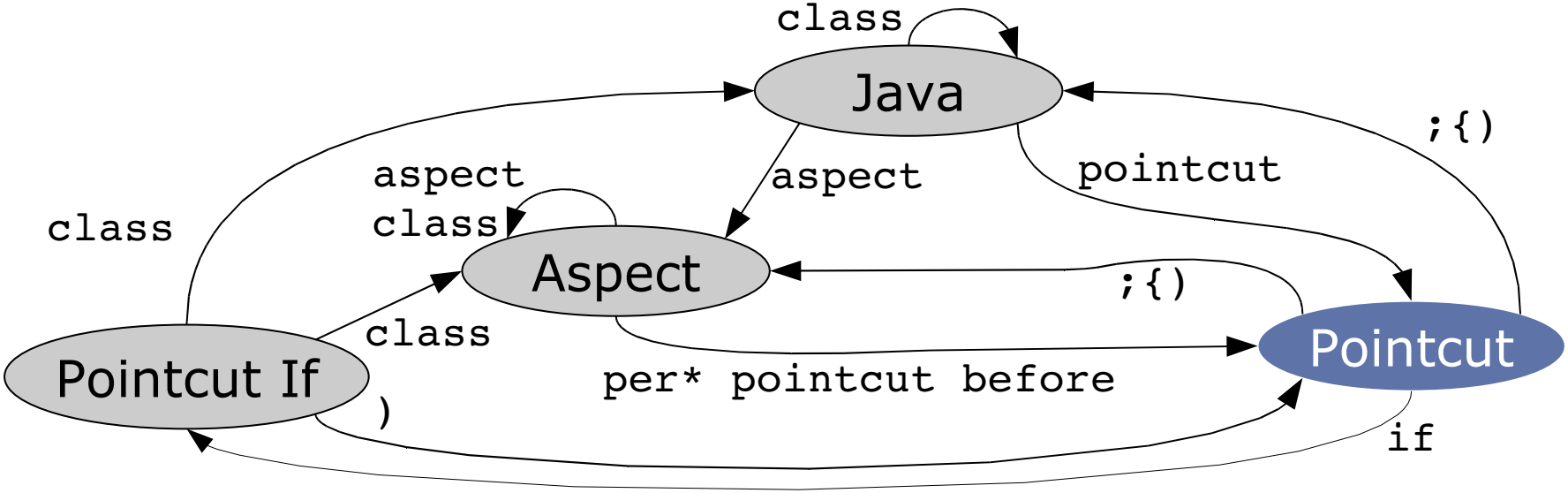
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



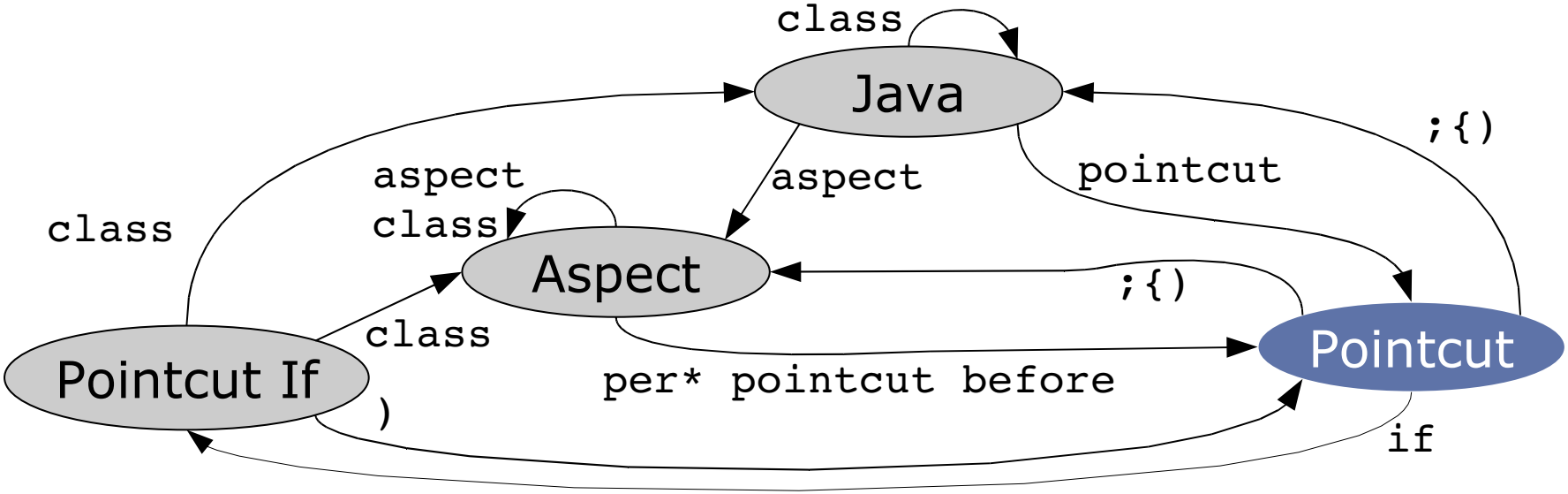
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



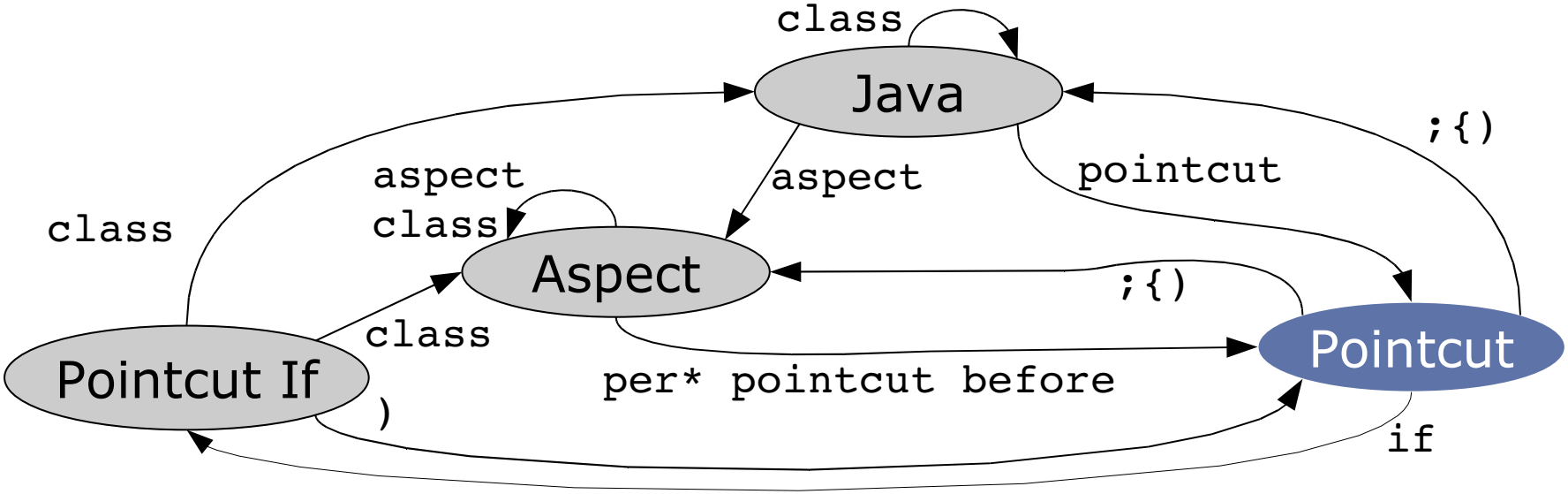
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



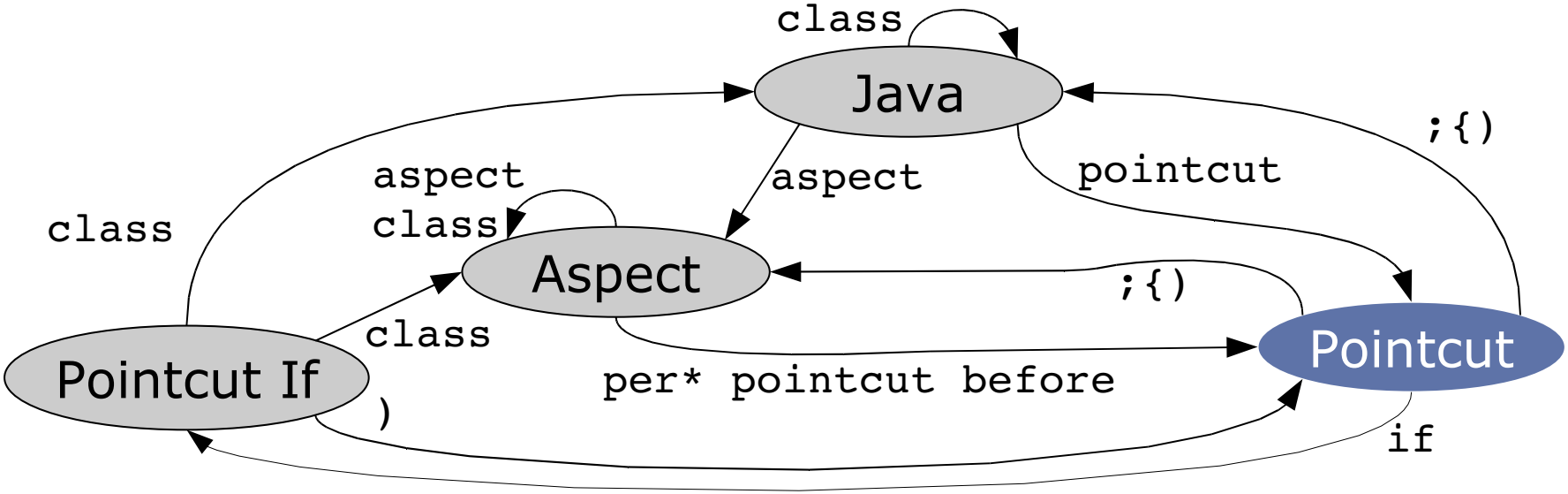
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



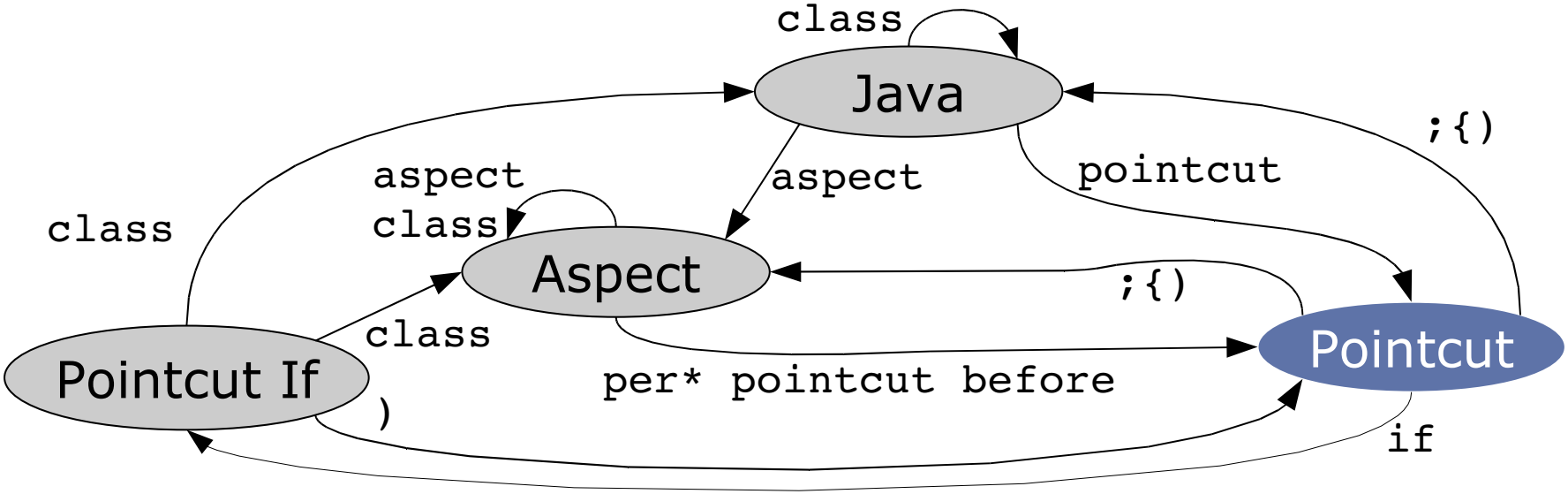
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



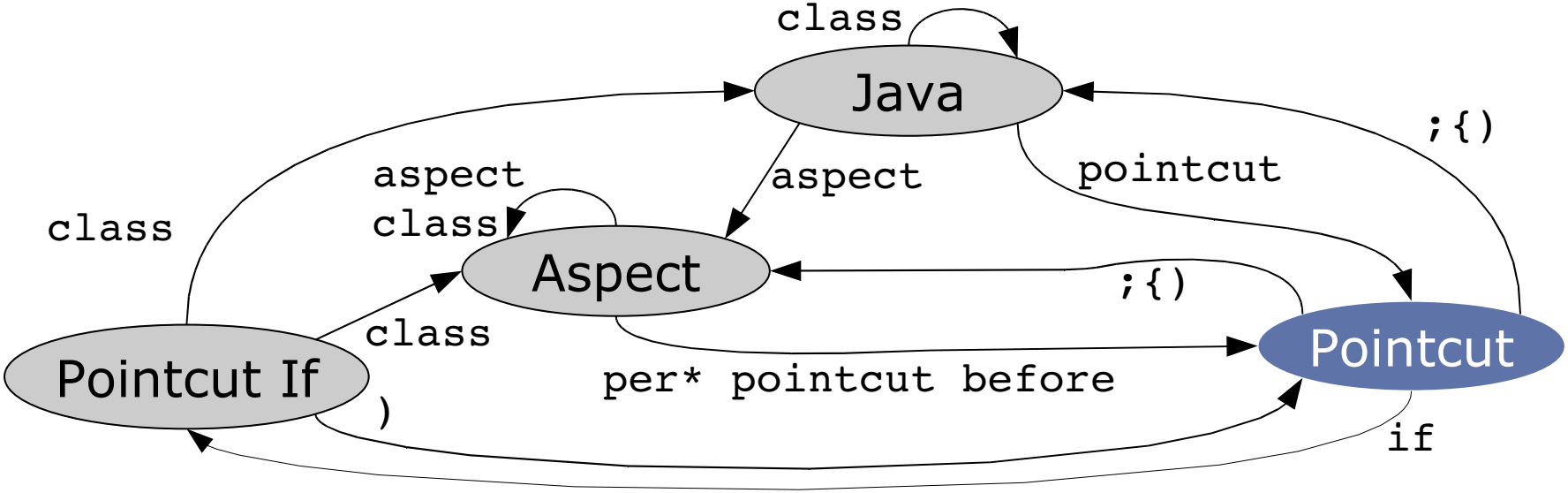
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



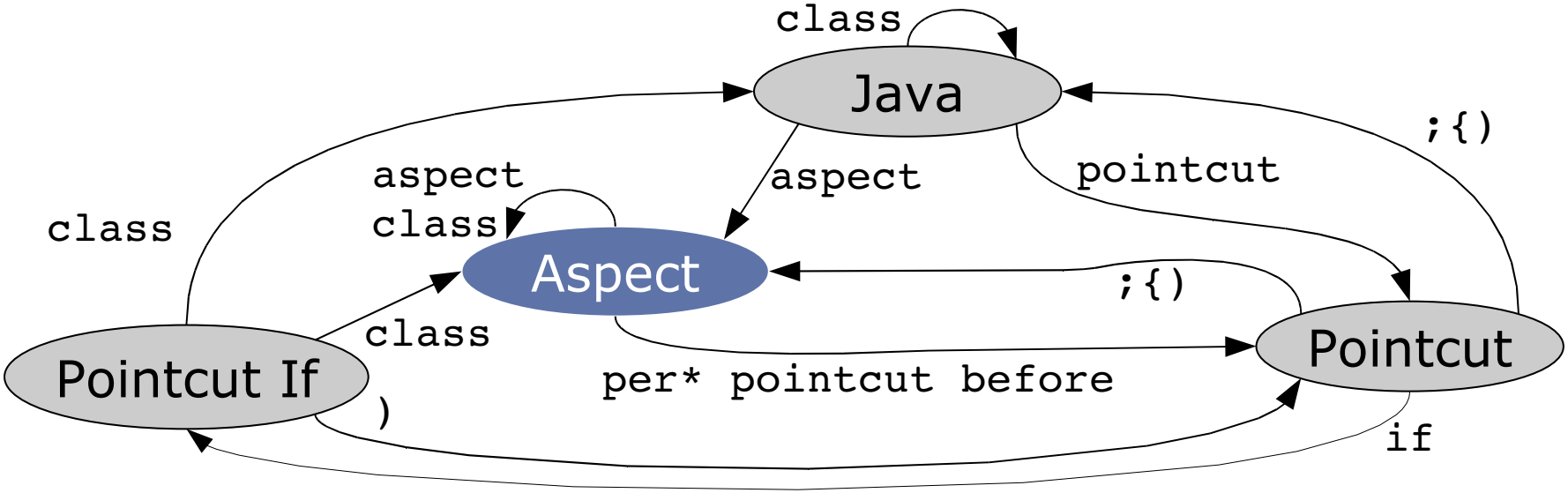
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



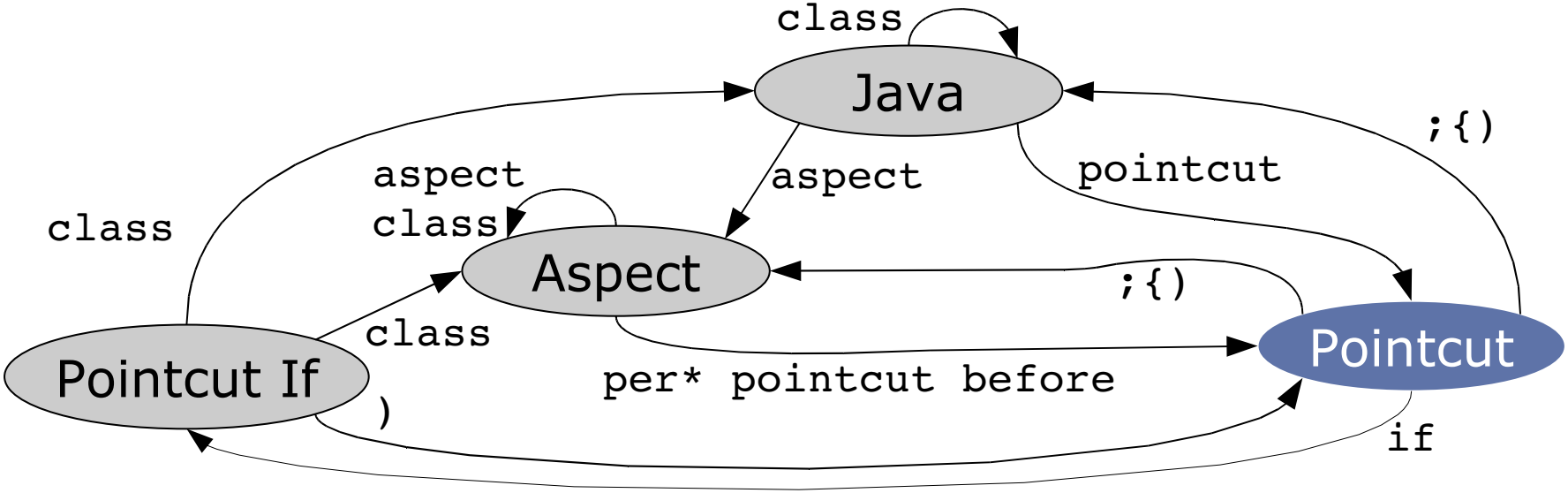
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



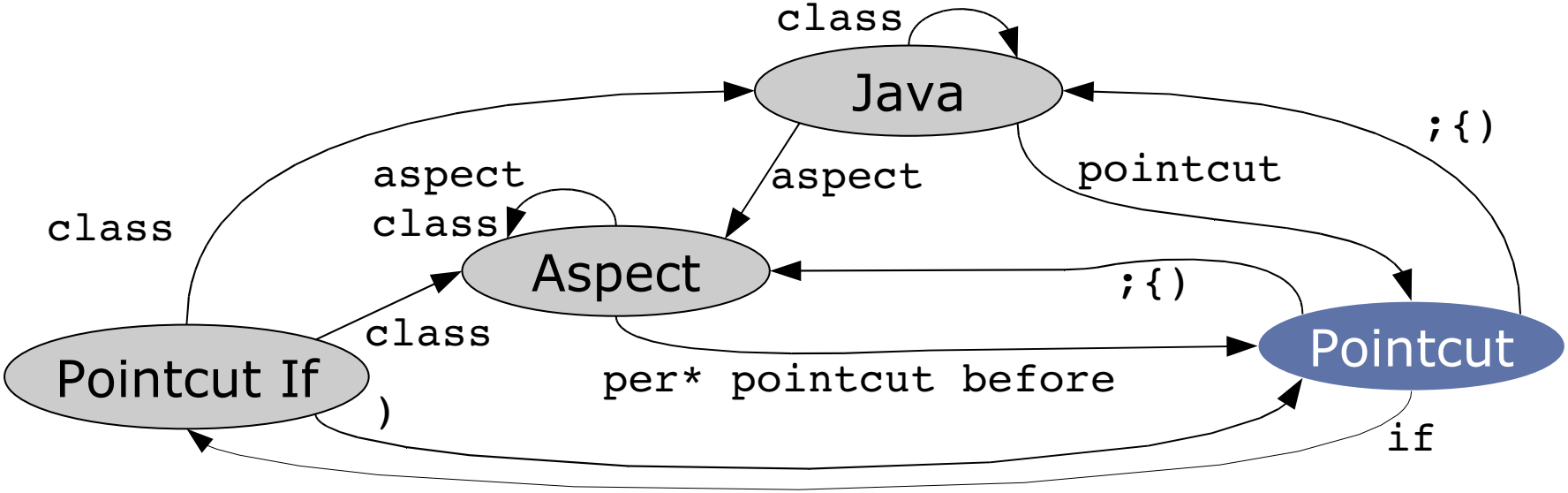
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



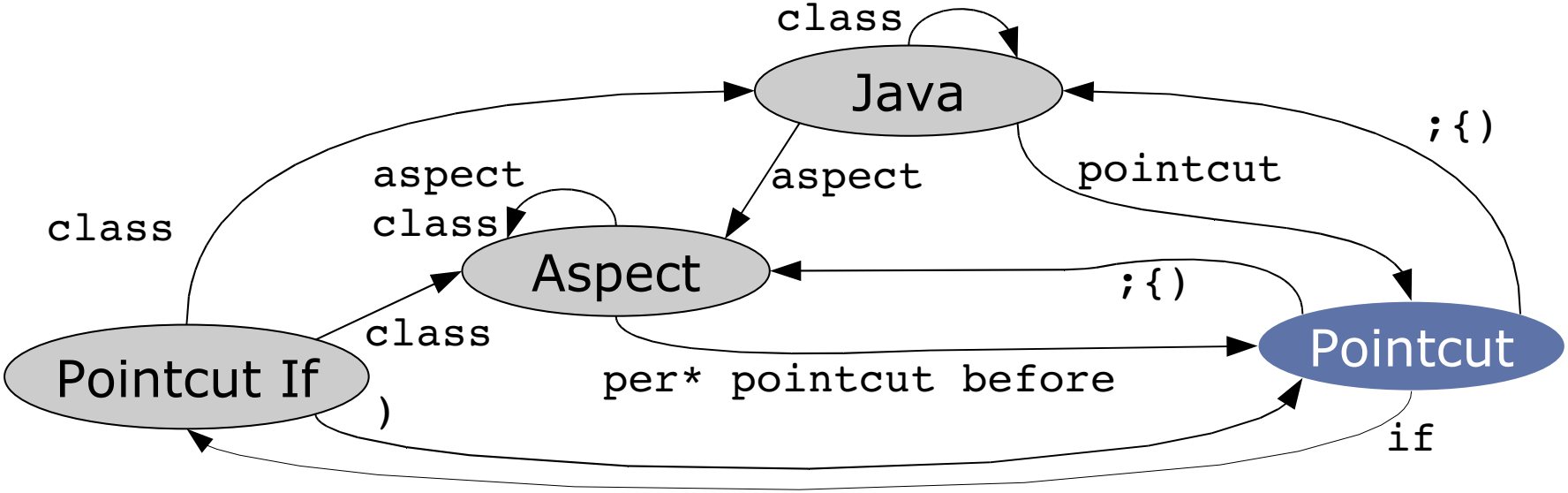
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



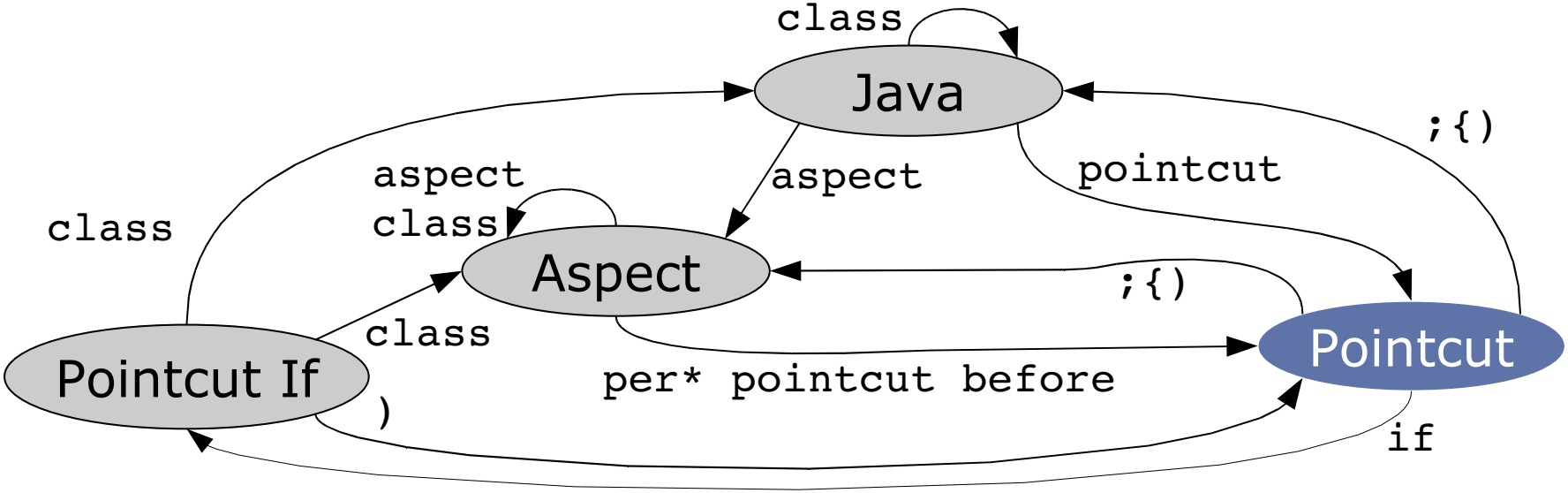
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



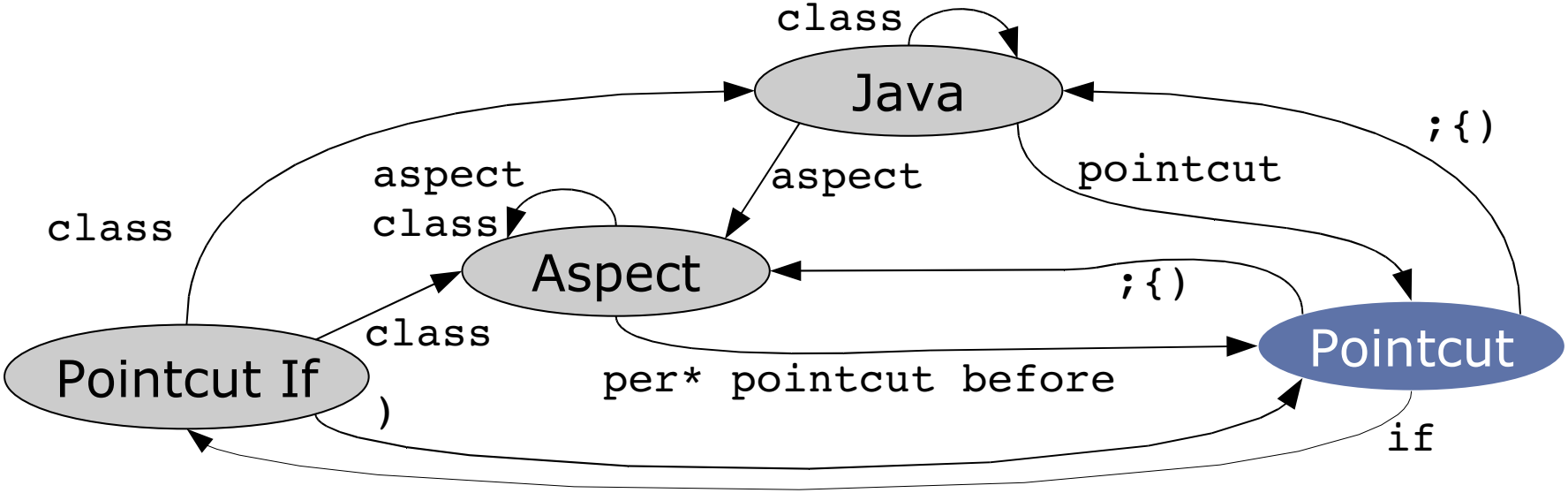
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



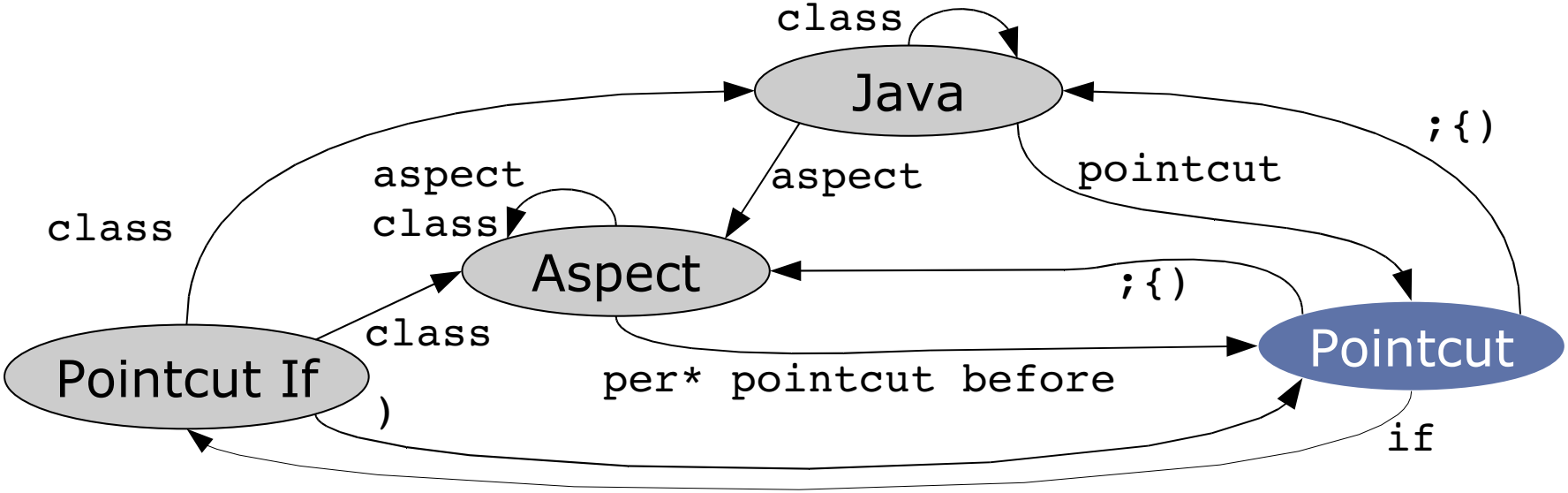
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



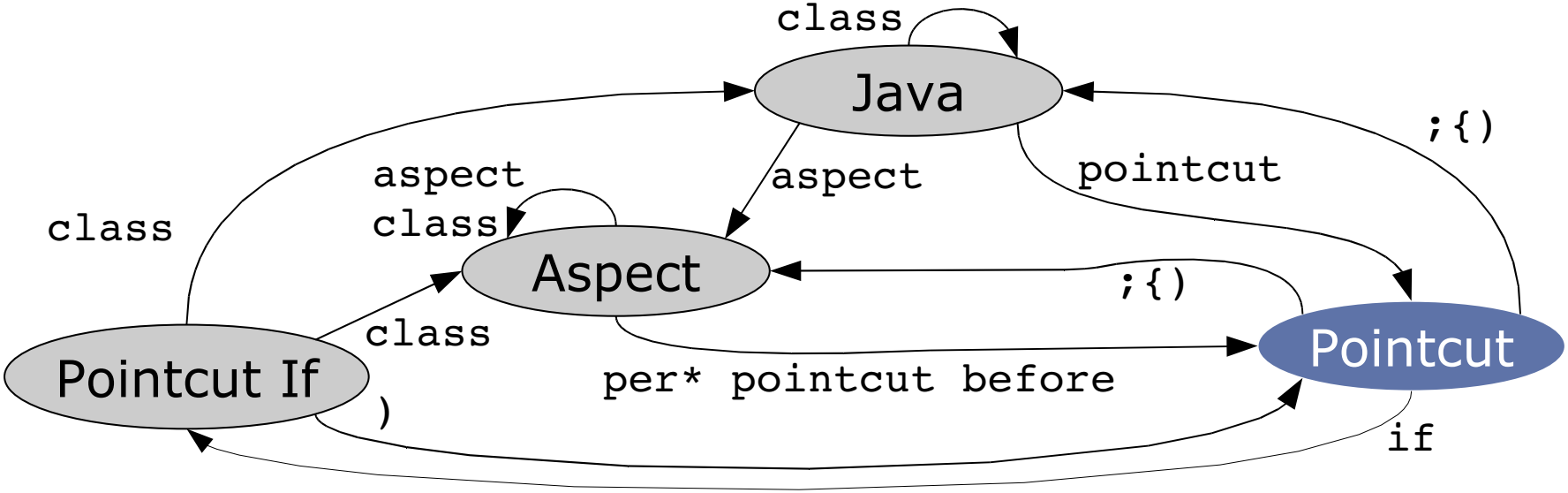
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



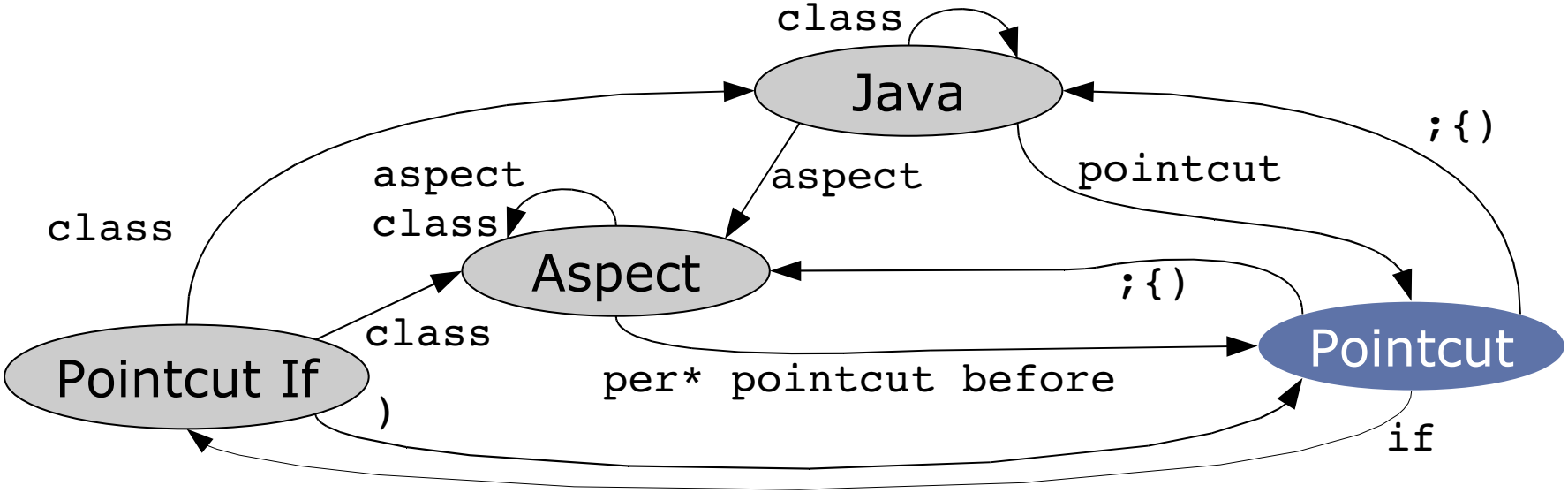
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



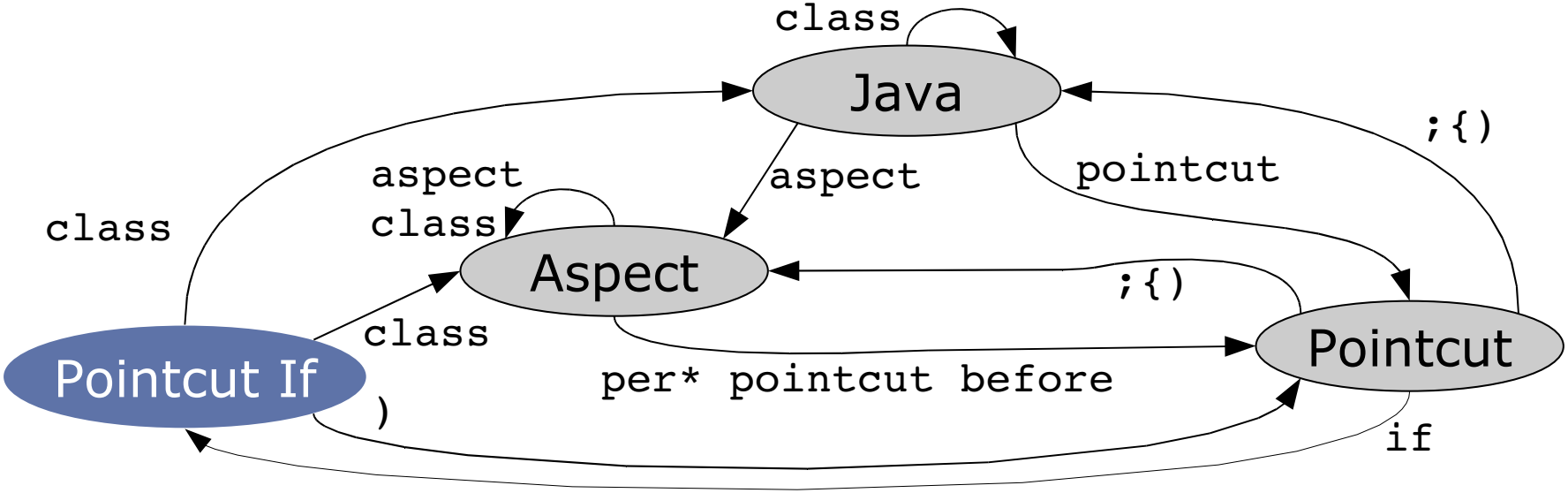
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



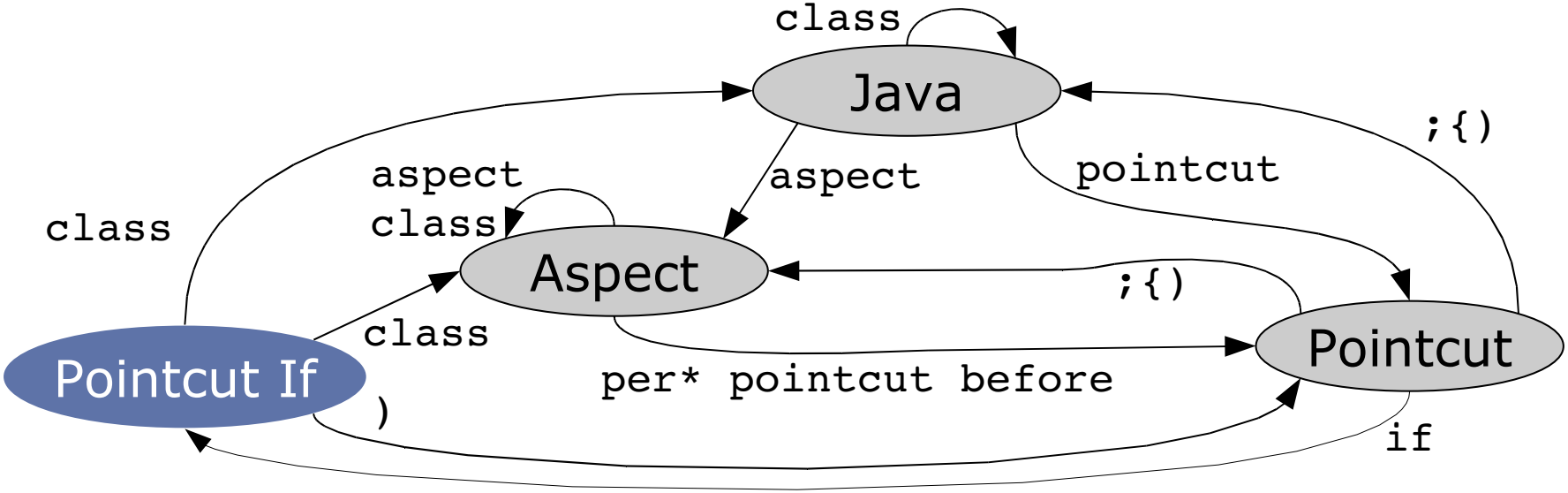
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



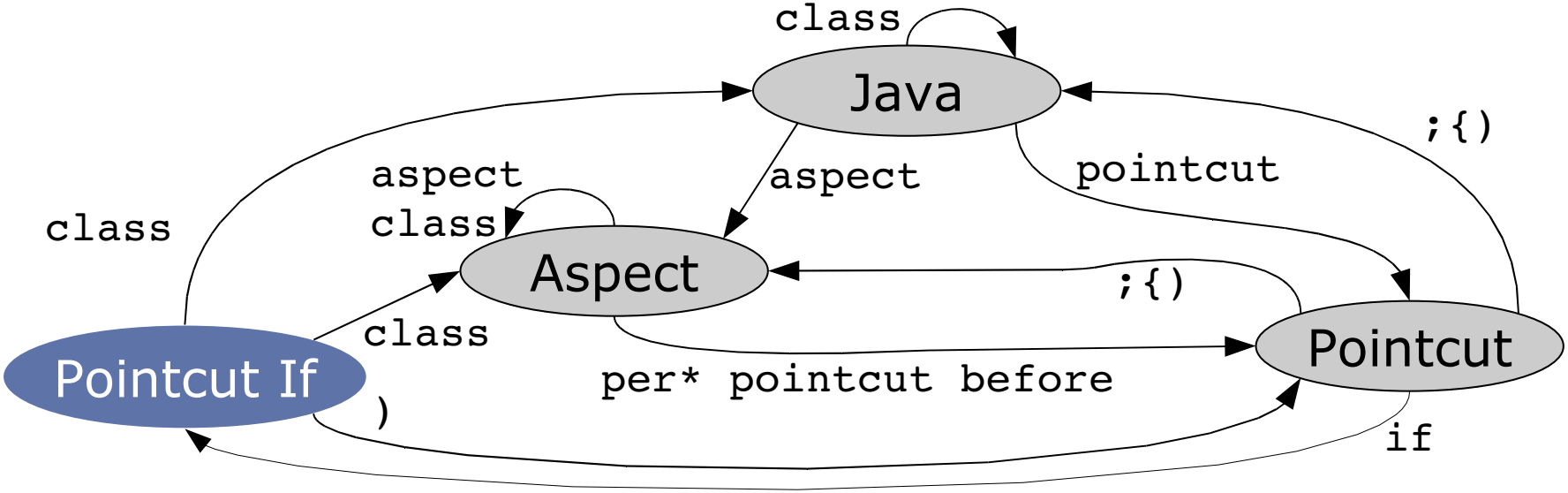
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



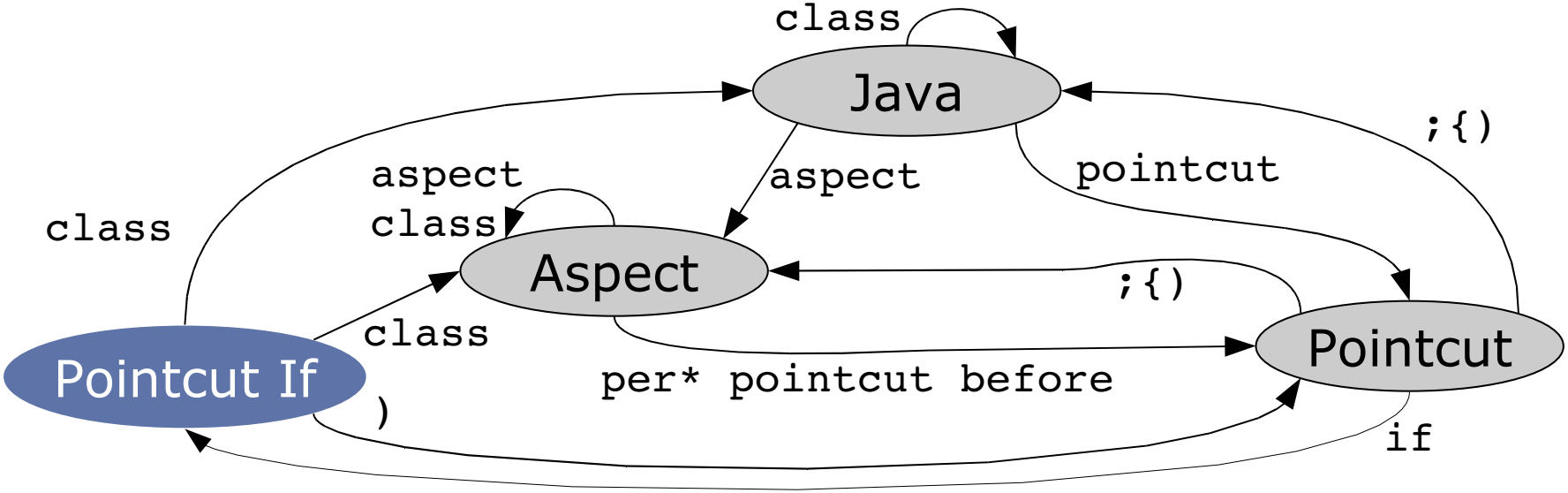
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



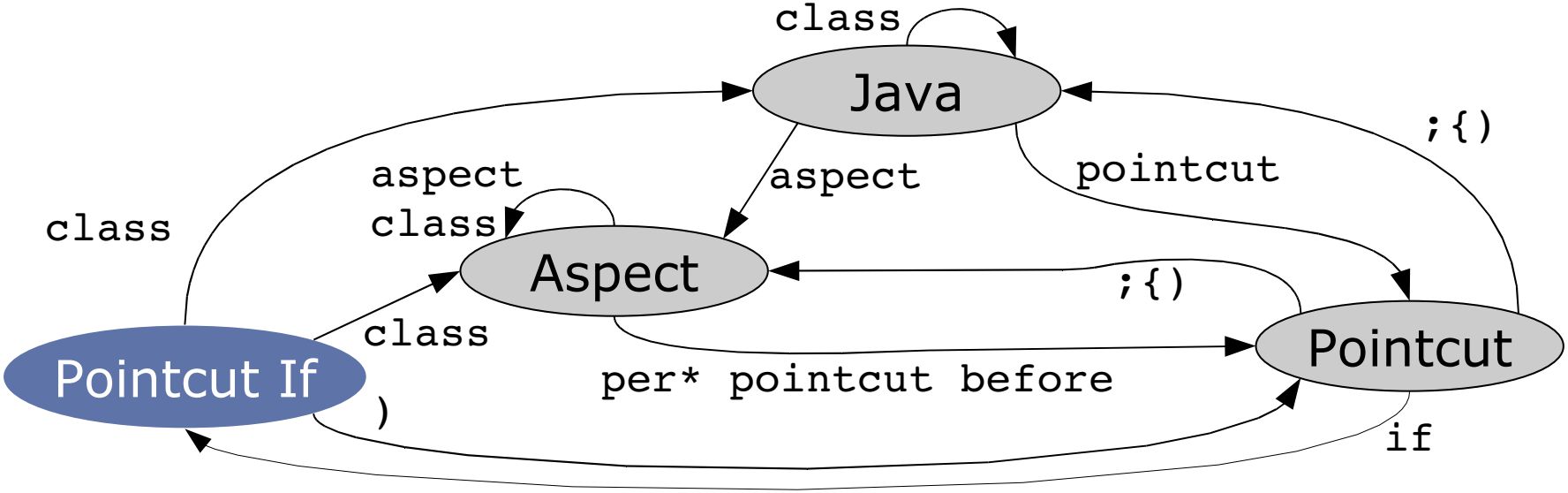
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



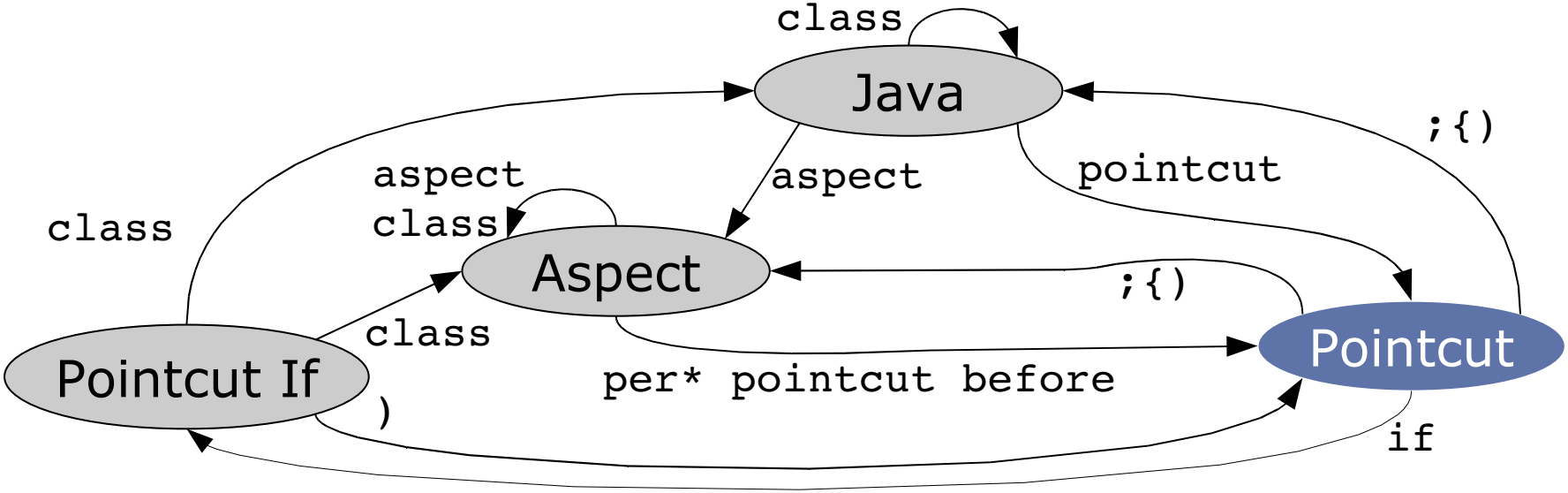
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



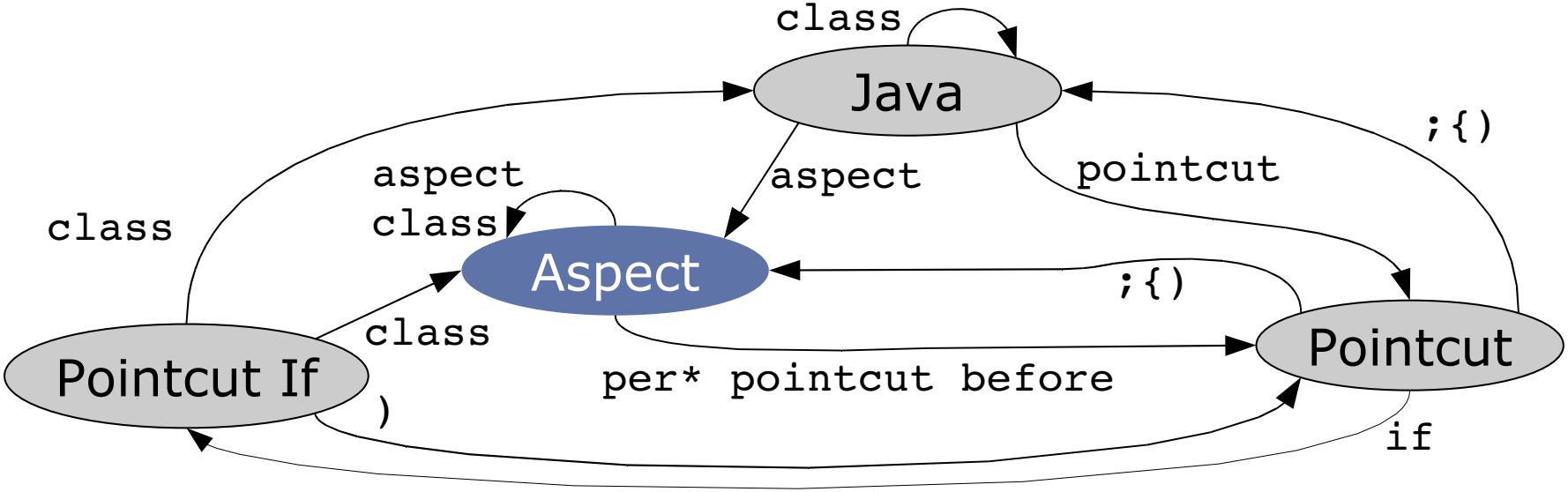
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



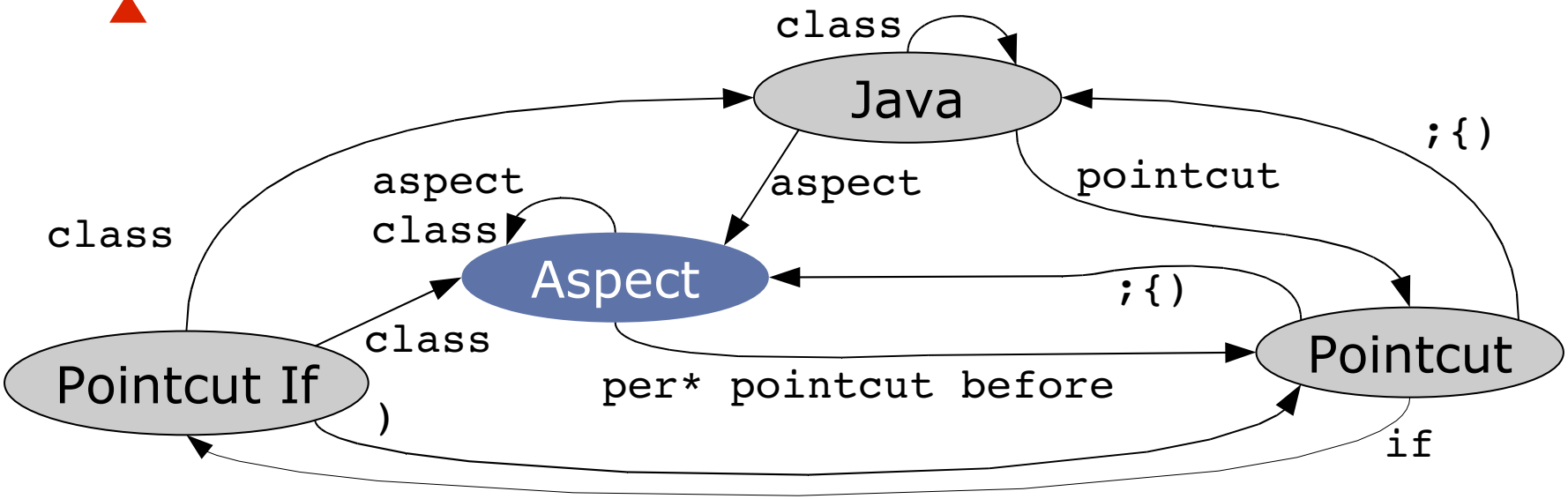
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



Keywords do not uniquely indicate next state.

- Type declaration: transition to Java or AspectJ

```
public class Foo {}
```

- Class literal: no transition

```
Class c = Foo.class;
```

Solution

```
lexer.addGlobalKeyword("class",
  new LexerAction_c(new Integer(sym.CLASS)) {
    public int getToken(AbcLexer lexer) {
      if(!lexer.getLastTokenWasDot())
        lexer.enterLexerState(aspectj or java);
      return token.intValue();
    }});
```

Managing lexical state in abc: class declarations 6

Keywords do not uniquely indicate next state.

- Type declaration: transition to Java or AspectJ

```
public class Foo {}
```

- Class literal: no transition

```
Class c = Foo.class;
```

Solution

```
lexer.addGlobalKeyword("class",
  new LexerAction_c(new Integer(sym.CLASS)) {
    public int getToken(AbcLexer lexer) {
      if(!lexer.getLastTokenWasDot())
        lexer.enterLexerState(aspectj or java);
      return token.intValue();
    }});
```

Keywords do not uniquely indicate next state.

- Type declaration: transition to Java or AspectJ

```
public class Foo {}
```

- Class literal: no transition

```
Class c = Foo.class;
```

Solution

```
lexer.addGlobalKeyword("class",
  new LexerAction_c(new Integer(sym.CLASS)) {
    public int getToken(AbcLexer lexer) {
      if(!lexer.getLastTokenWasDot())
        lexer.enterLexerState(aspectj or java);
      return token.intValue();
    }});
```

Summary of ajc and abc

Approaches

- ajc: multiple parsers, single scanner
- abc: single parser, multiple scanners (stateful)

Issues

- syntax defined operationally in scanner and parser
- no declarative formalization of AspectJ syntax
- various implementation quircks
- correctness of stateful scanner fragile
- context-free parsing in scanner duplicates work
- lexical states influenced by complexity of context-free parsing

Scannerless parsing elegantly deals with these issues

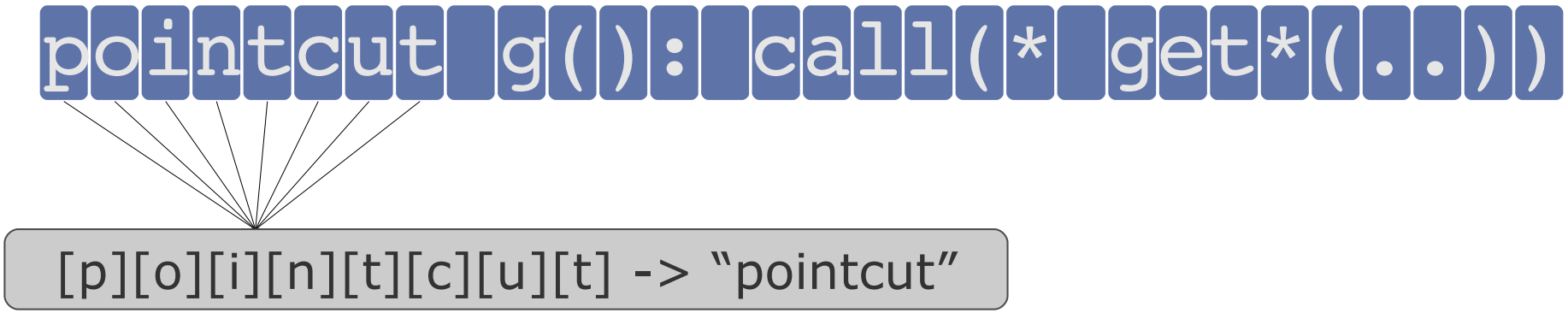
- no separate lexical analysis
- parser operates on individual characters
- scannerless parsing needs generalized-LR parsing

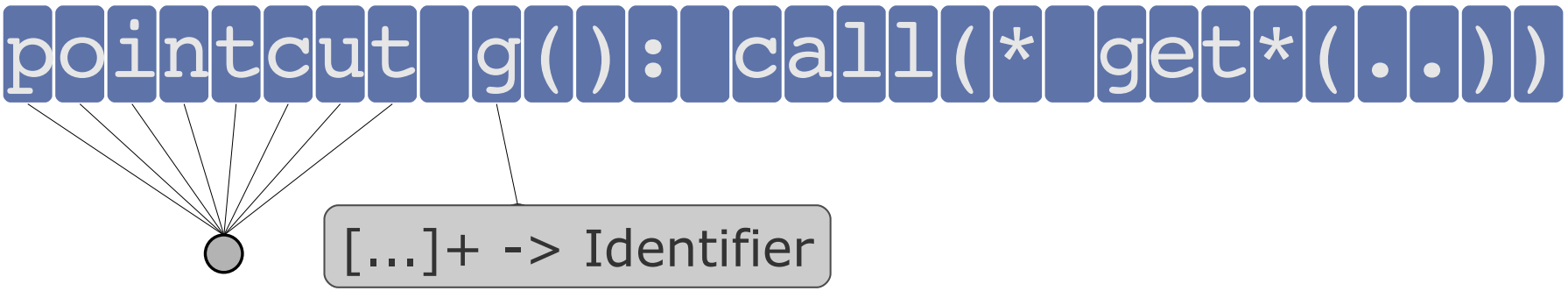
SDF – Syntax Definition Formalism

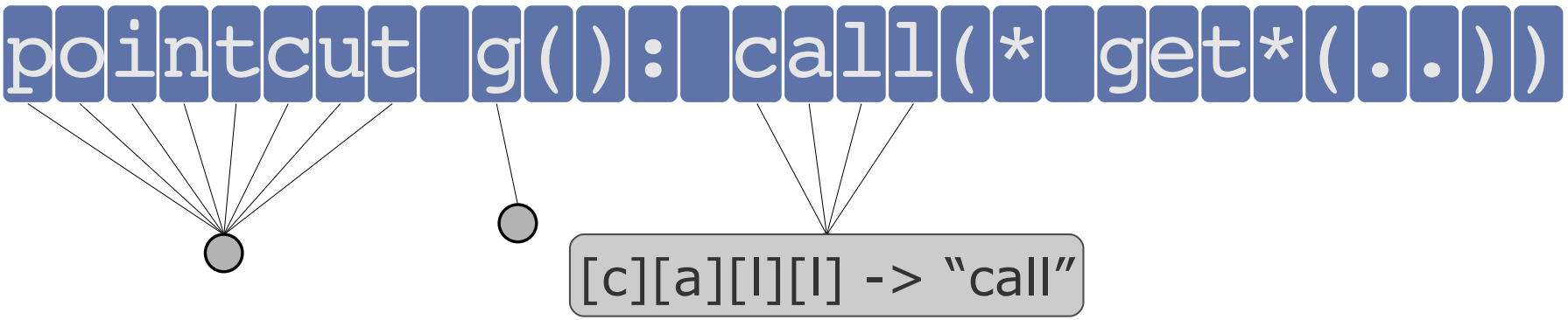
1. declarative
2. modular
3. context-free and lexical syntax
4. declarative disambiguation
5. all context-free grammars

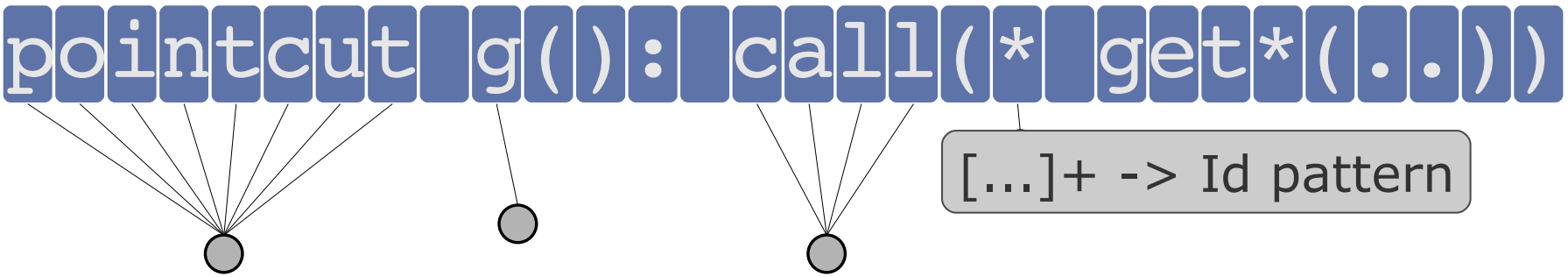
```
pointcut g(): call(* get*(..))
```

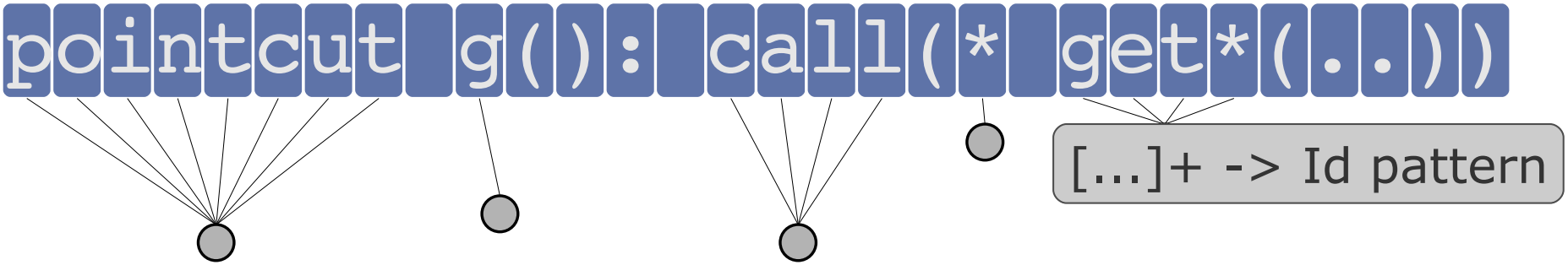
```
pointcut g(): call(* get*(..))
```

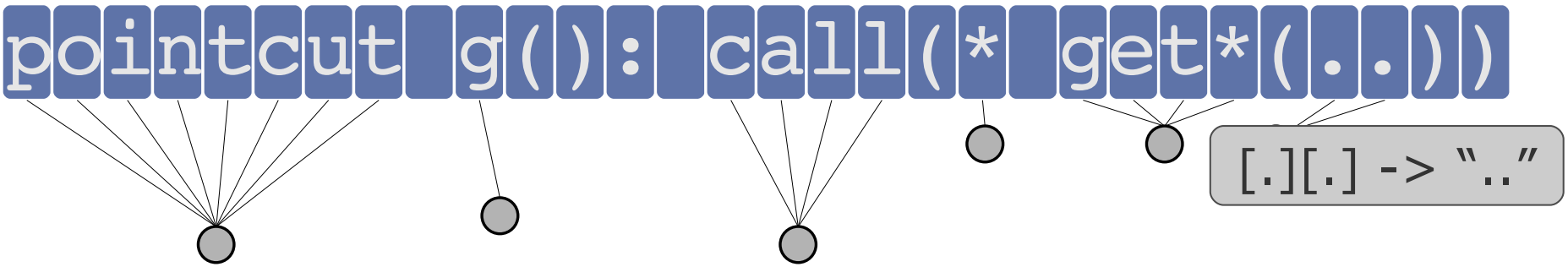

pointcut g(): call(* get*(..))
[p][o][i][n][t][c][u][t] -> “pointcut”

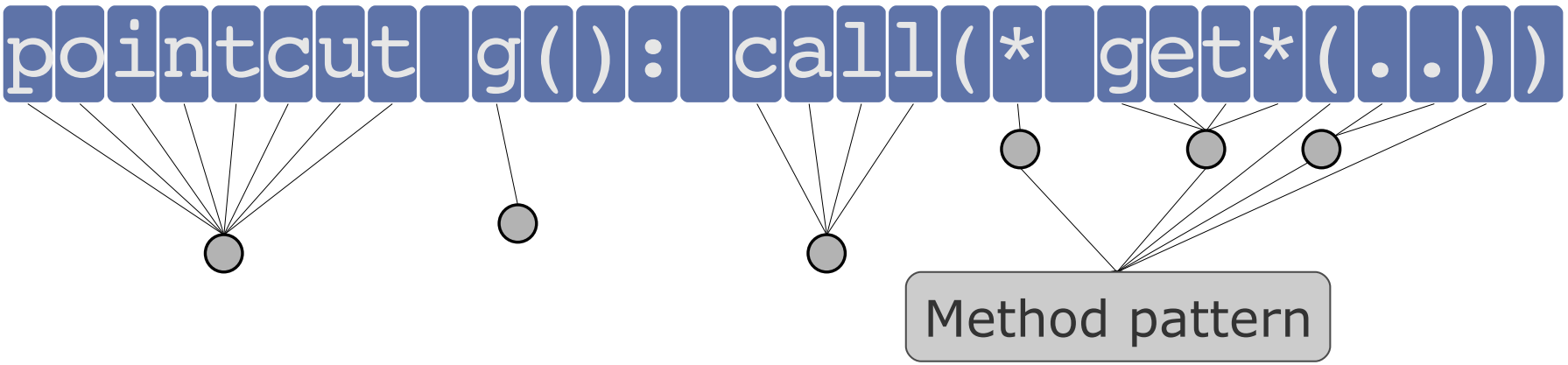

pointcut g(): call(* get*(..))
[...]+ -> Identifier

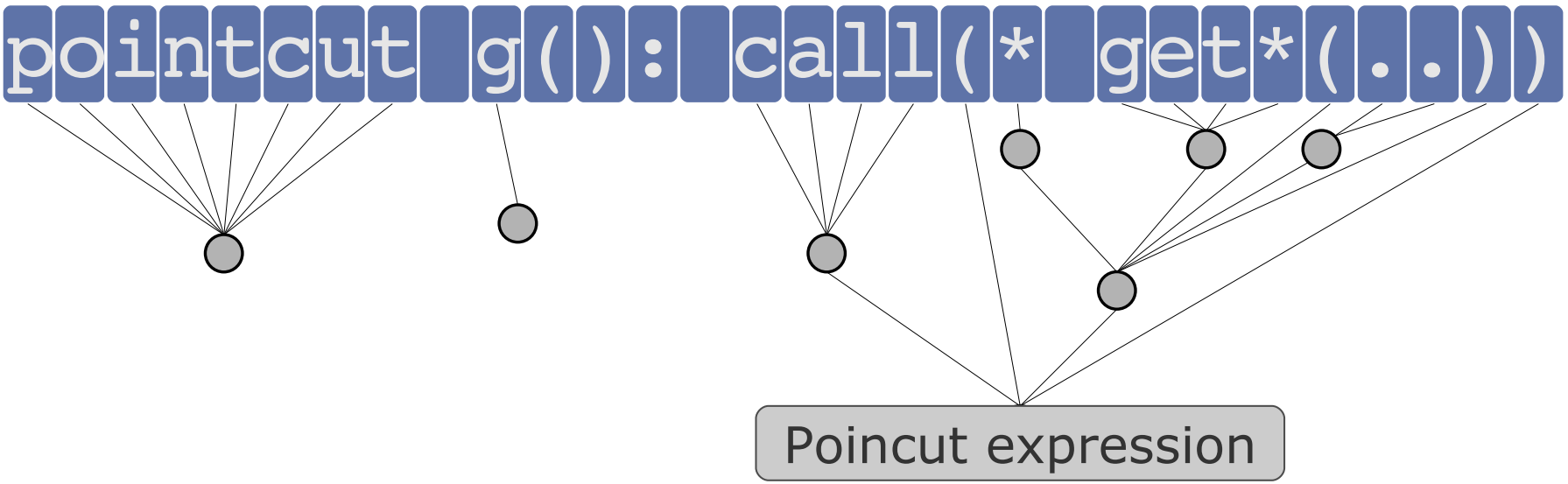

pointcut g(): call(* get*(..))
[c][a][l][l] -> “call”

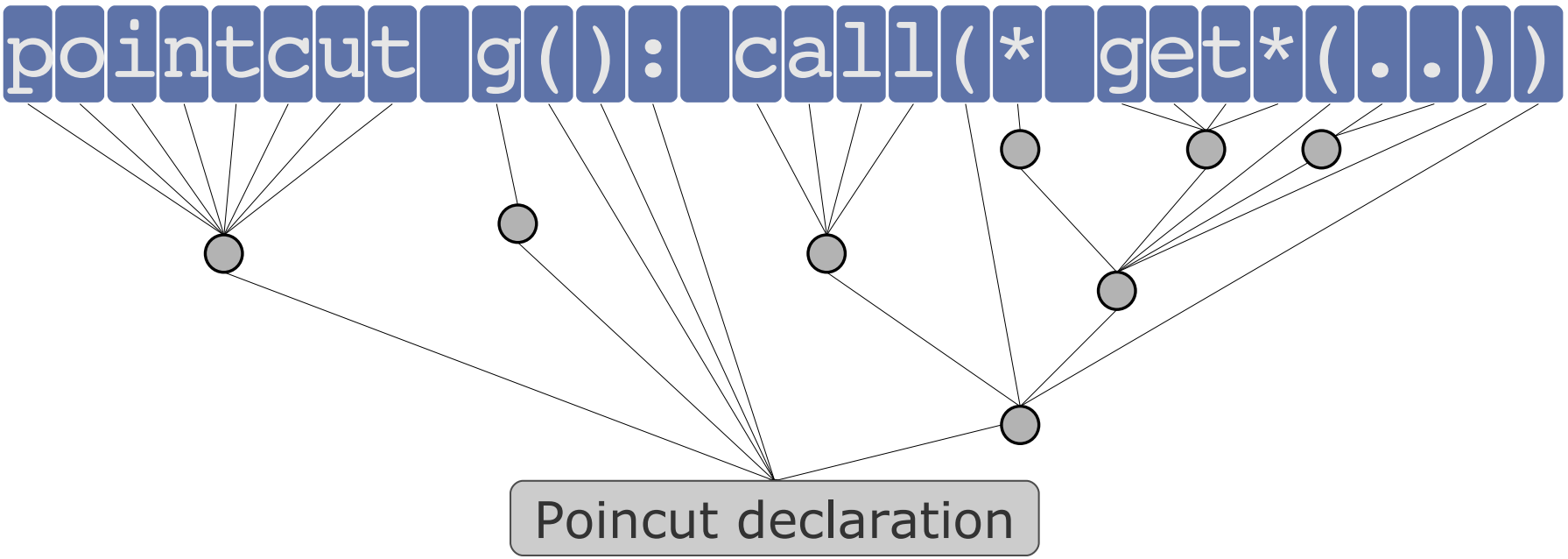

pointcut g(): call(* get*(..))
[...]+ -> Id pattern


pointcut g(): call(* get*(..))
[...]+ -> Id pattern


pointcut g(): call(* get*(..))
[.][.] -> ".."


pointcut g(): call(* get*(..))
Method pattern


pointcut g(): call(* get*(..))
Poincut expression


pointcut g(): call(* get*(..))
Poincut declaration

```
module Java imports Statements Expressions Identifiers
exports context-free syntax
  PackageDec? ImportDec* TypeDec+ -> CompilationUnit
```

```
module Statements exports context-free syntax
  "for" "(" FormalParam ":" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions exports context-free syntax
  ExprName                          -> Expr
  Expr "+" Expr                     -> Expr {left}
  MethodSpec "(" {Expr ","}* ")"    -> Expr
  MethodName                        -> MethodSpec
```

```
module Identifiers exports lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id
```

```
module Java imports Statements Expressions Identifiers
exports context-free syntax
  PackageDec? ImportDec* TypeDec+ -> CompilationUnit
```

```
module Statements exports context-free syntax
  "for" "(" FormalParam ":" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions exports context-free syntax
  ExprName                          -> Expr
  Expr "+" Expr                     -> Expr {left}
  MethodSpec "(" {Expr ","}* ")"    -> Expr
  MethodName                        -> MethodSpec
```

```
module Identifiers exports lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id
```

```
module Java imports Statements Expressions Identifiers
exports context-free syntax
  PackageDec? ImportDec* TypeDec+ -> CompilationUnit
```

```
module Statements exports context-free syntax
  "for" "(" FormalParam ":" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions exports context-free syntax
  ExprName                          -> Expr
  Expr "+" Expr                     -> Expr {left}
  MethodSpec "(" {Expr ","}* ")"    -> Expr
  MethodName                        -> MethodSpec
```

```
module Identifiers exports lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id
```

```
module Java imports Statements Expressions Identifiers
exports context-free syntax
  PackageDec? ImportDec* TypeDec+ -> CompilationUnit
```

```
module Statements exports context-free syntax
  "for" "(" FormalParam ":" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions exports context-free syntax
  ExprName                          -> Expr
  Expr "+" Expr                     -> Expr {left}
  MethodSpec "(" {Expr ","}* ")"    -> Expr
  MethodName                        -> MethodSpec
```

```
module Identifiers exports lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id
```

```
module Java imports Statements Expressions Identifiers
exports context-free syntax
  PackageDec? ImportDec* TypeDec+ -> CompilationUnit
```

```
module Statements exports context-free syntax
  "for" "(" FormalParam ":" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions exports context-free syntax
  ExprName                          -> Expr
  Expr "+" Expr                     -> Expr {left}
  MethodSpec "(" {Expr ","}* ")"    -> Expr
  MethodName                        -> MethodSpec
```

```
module Identifiers exports lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

```
module AspectDeclaration exports context-free syntax
  AdvMod* AdvSpec Throws? ":" PointcutExpr MethodBody -> AdviceDec
  "before" "(" {Param ","}* ")" -> AdvSpec
  ResultType "around" "(" {Param ","}* ")"  -> AdvSpec
```

```
module PointcutExpression exports context-free syntax
  "call"  "(" MethodConstrPattern ")" -> PointcutExpr
  "cflow" "(" PointcutExpr ")"        -> PointcutExpr
  "if"    "(" Expr ")"                -> PointcutExpr
```

```
module Pattern exports
context-free syntax
  IdPattern                   -> NamePattern
  NamePattern "."  IdPattern -> NamePattern
  NamePattern ".." IdPattern -> NamePattern
lexical syntax
  [A-Za-z\_\$\*][A-Za-z0-9\_\$\*]* -> IdPattern
```

- import modules of aspects, pointcuts, patterns
- compose the involved languages

```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

- import modules of aspects, pointcuts, patterns
- compose the involved languages

```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

- import modules of aspects, pointcuts, patterns
- compose the involved languages

```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

- import modules of aspects, pointcuts, patterns
- compose the involved languages

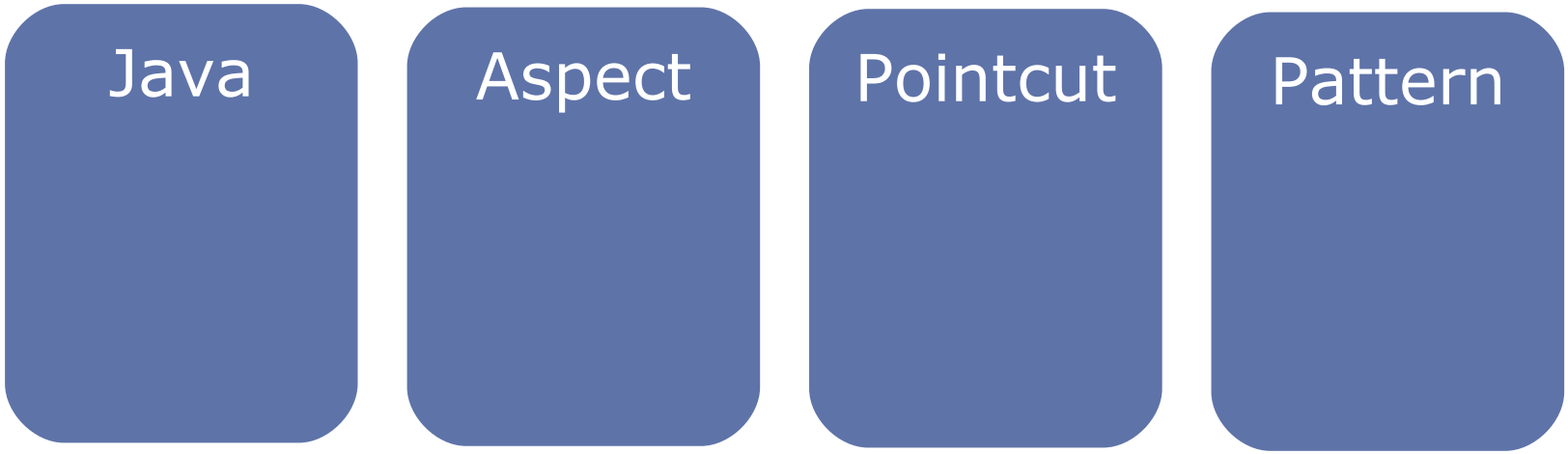
```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

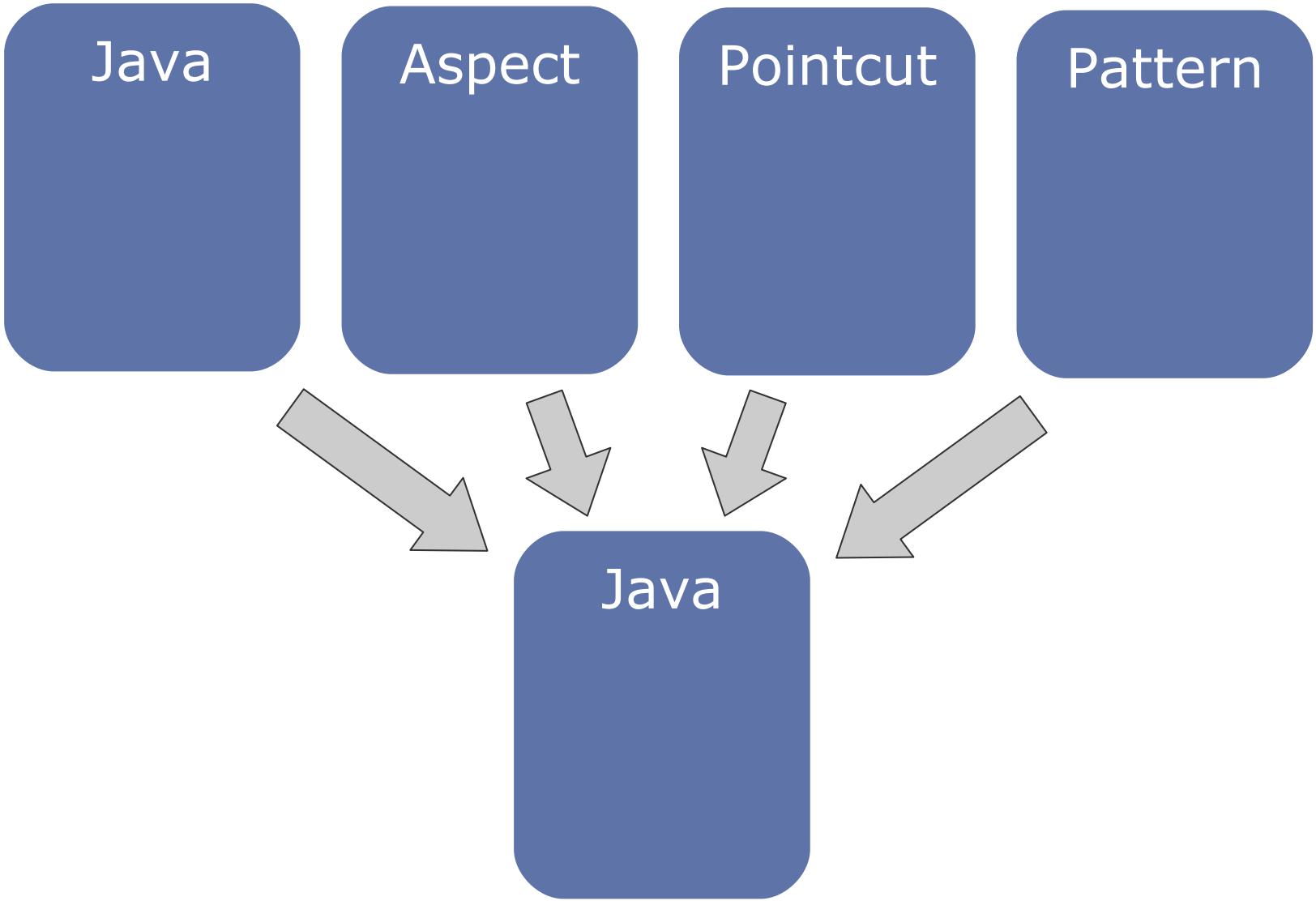
- import modules of aspects, pointcuts, patterns
- compose the involved languages

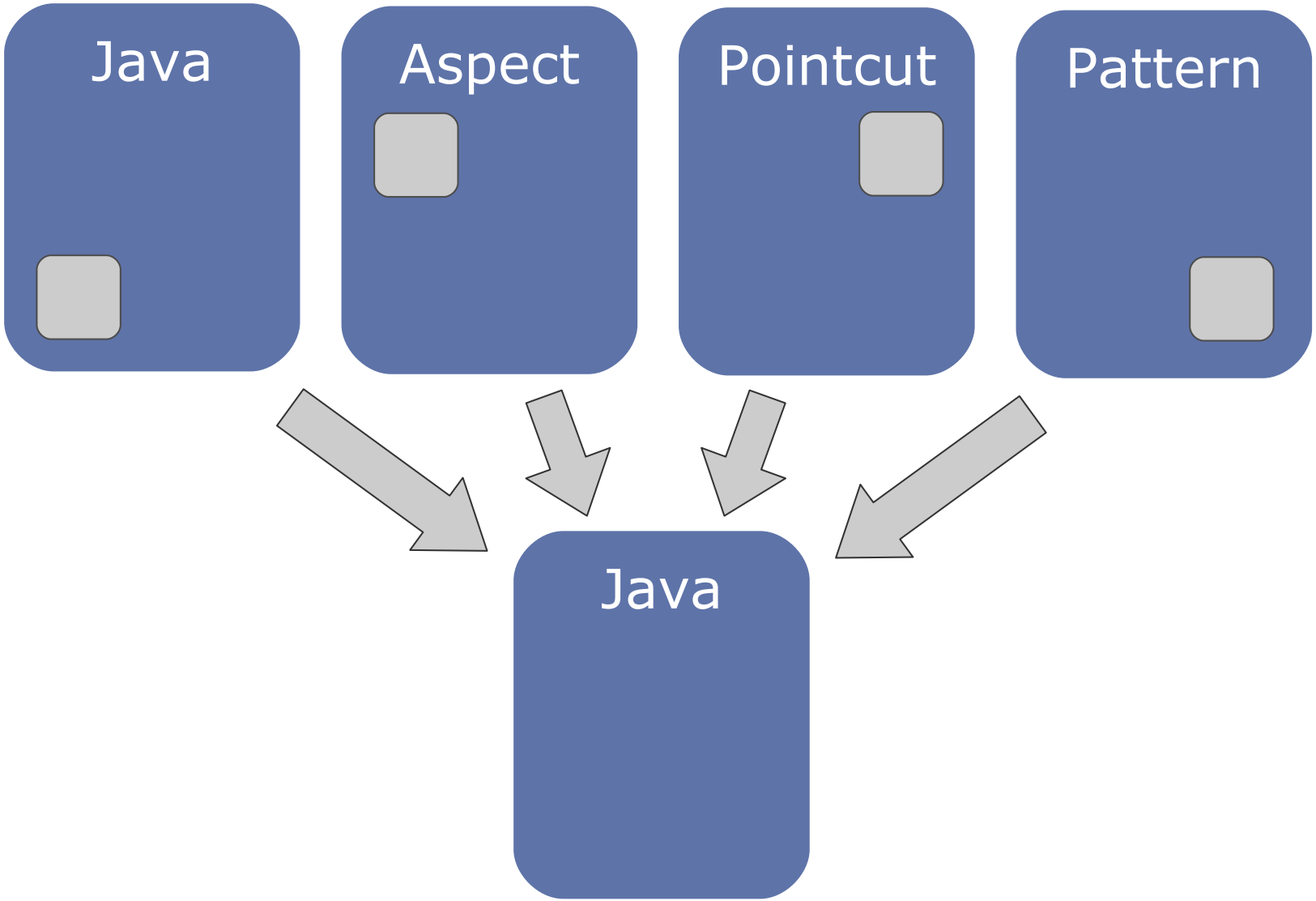
```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

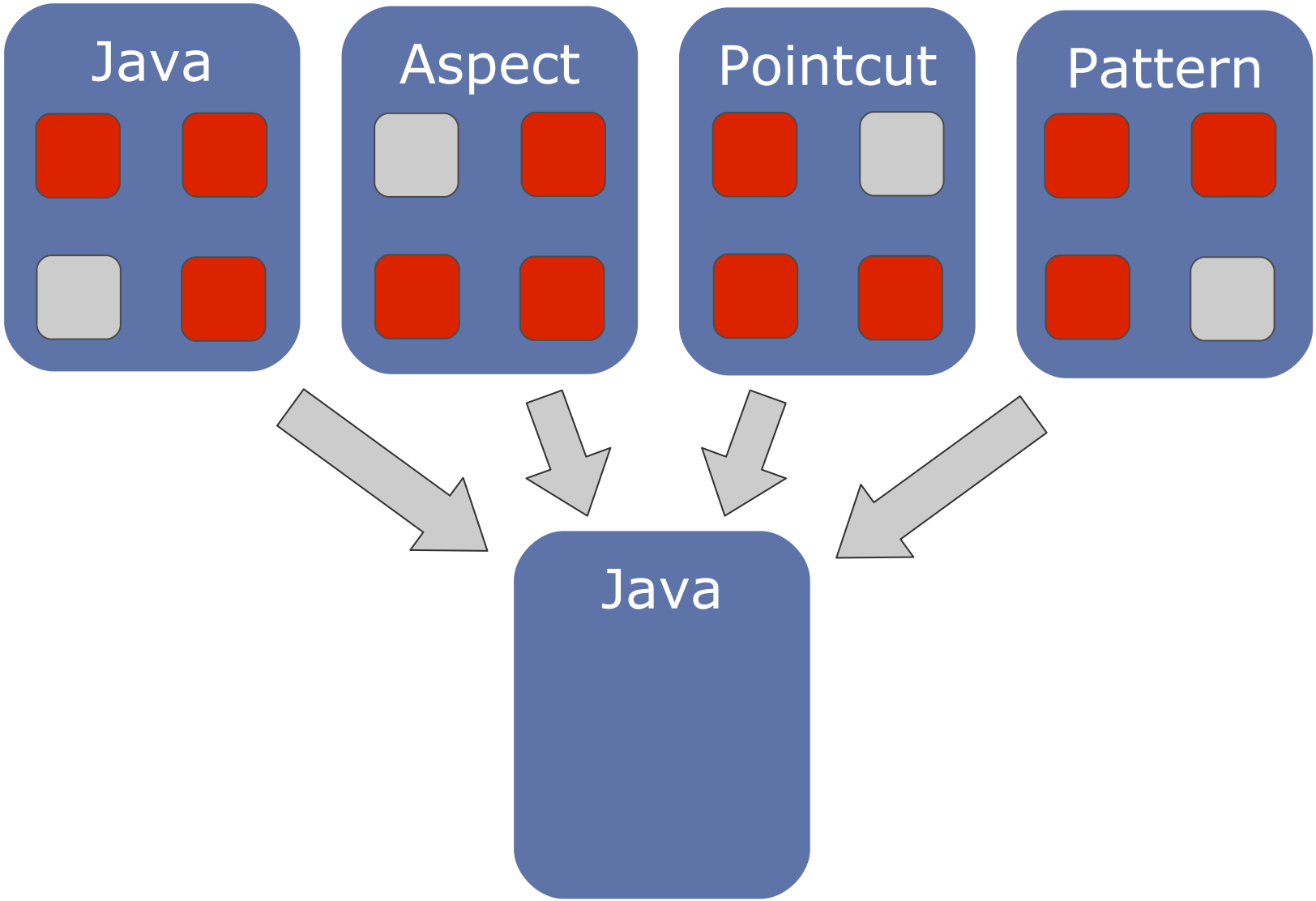
- import modules of aspects, pointcuts, patterns
- compose the involved languages

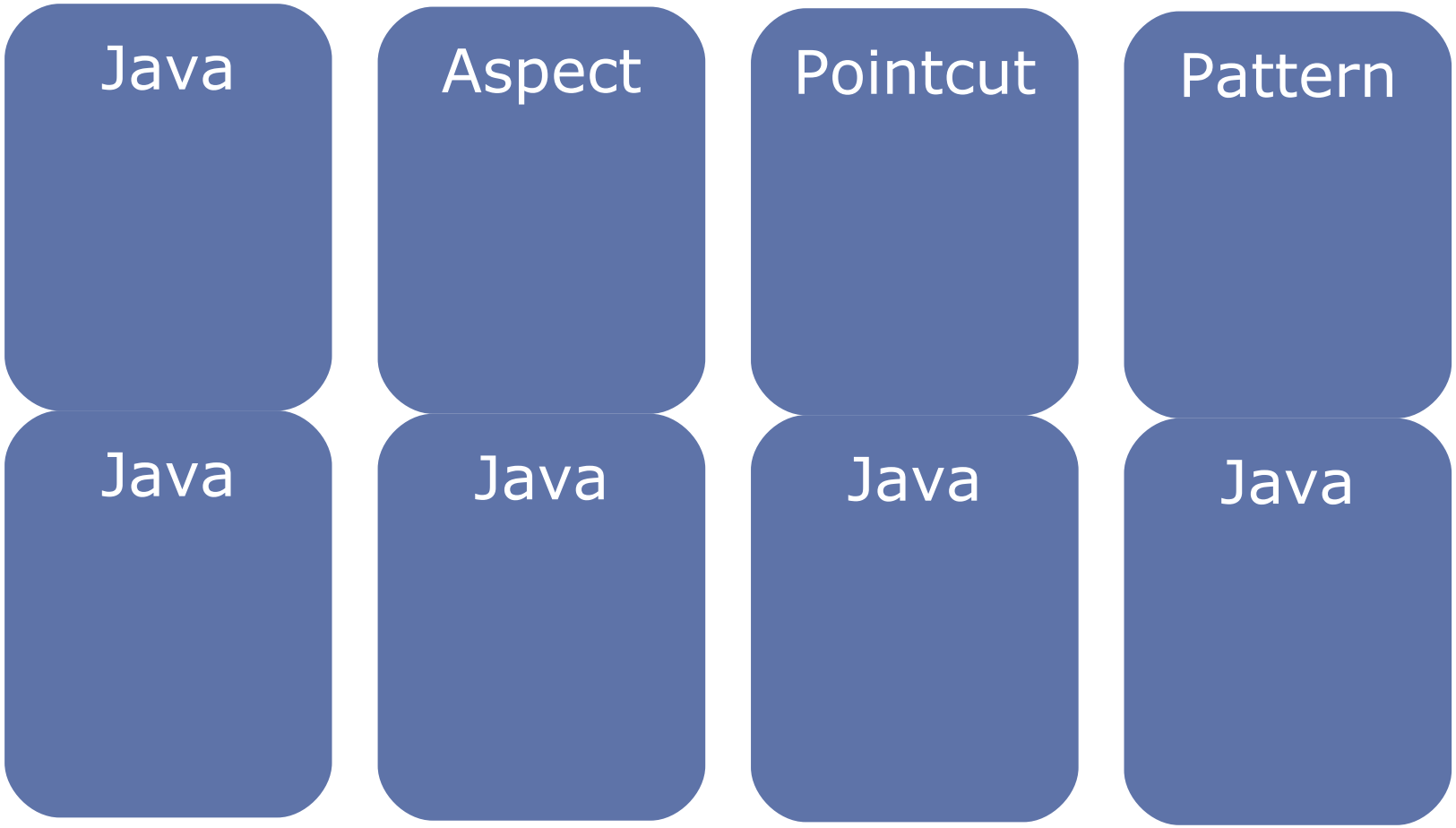
```
module AspectJ
imports Java AspectDeclaration PointcutExpression Pattern
exports context-free syntax
    AspectDec    -> TypeDec
    ClassBodyDec -> AspectBodyDec

    AspectDec    -> ClassMemberDec
    PointcutDec  -> ClassMemberDec

    AspectDec   -> InterfaceMemberDec
    PointcutDec -> InterfaceMemberDec
```

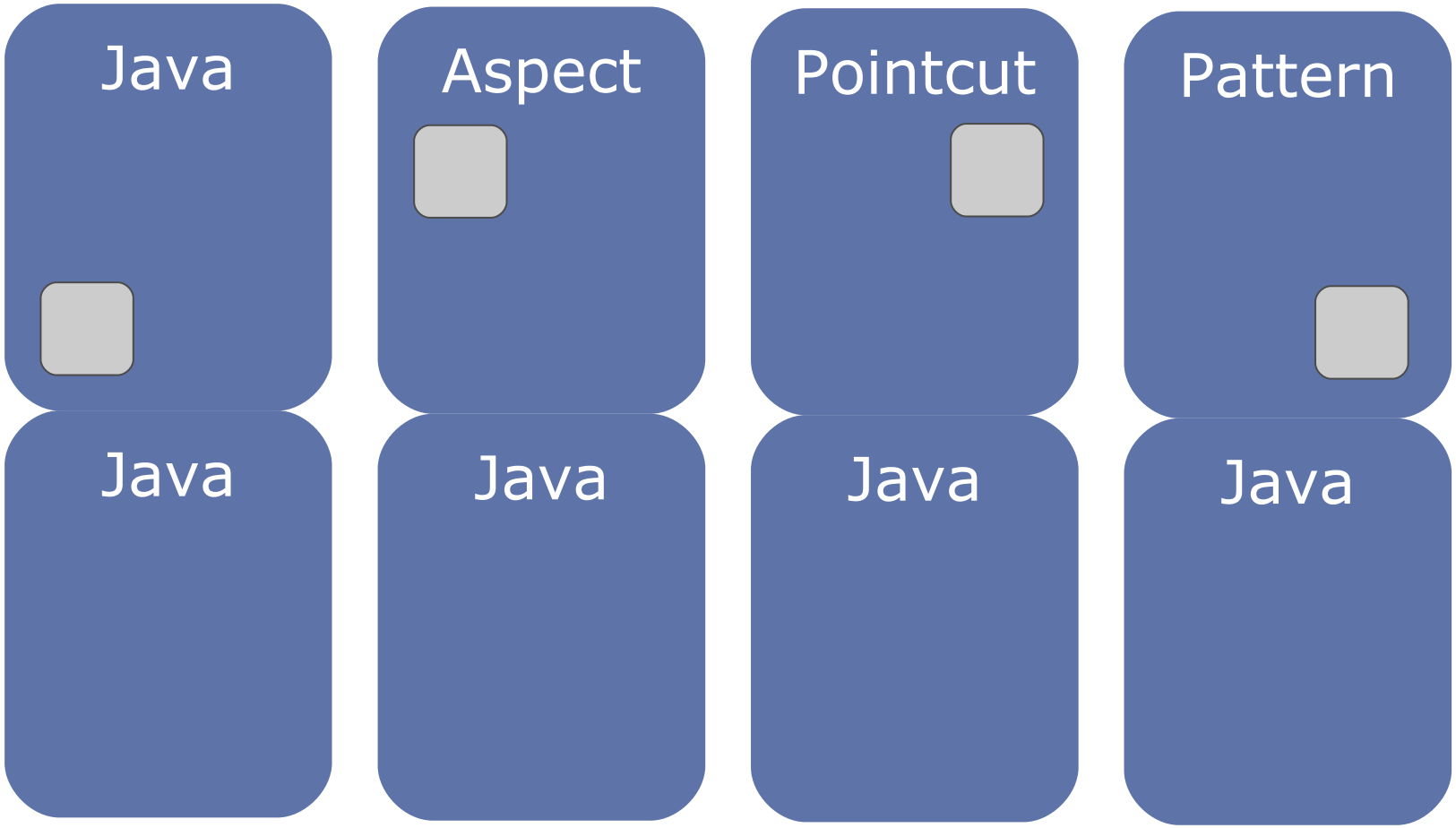

Java
Aspect
Pointcut
Pattern

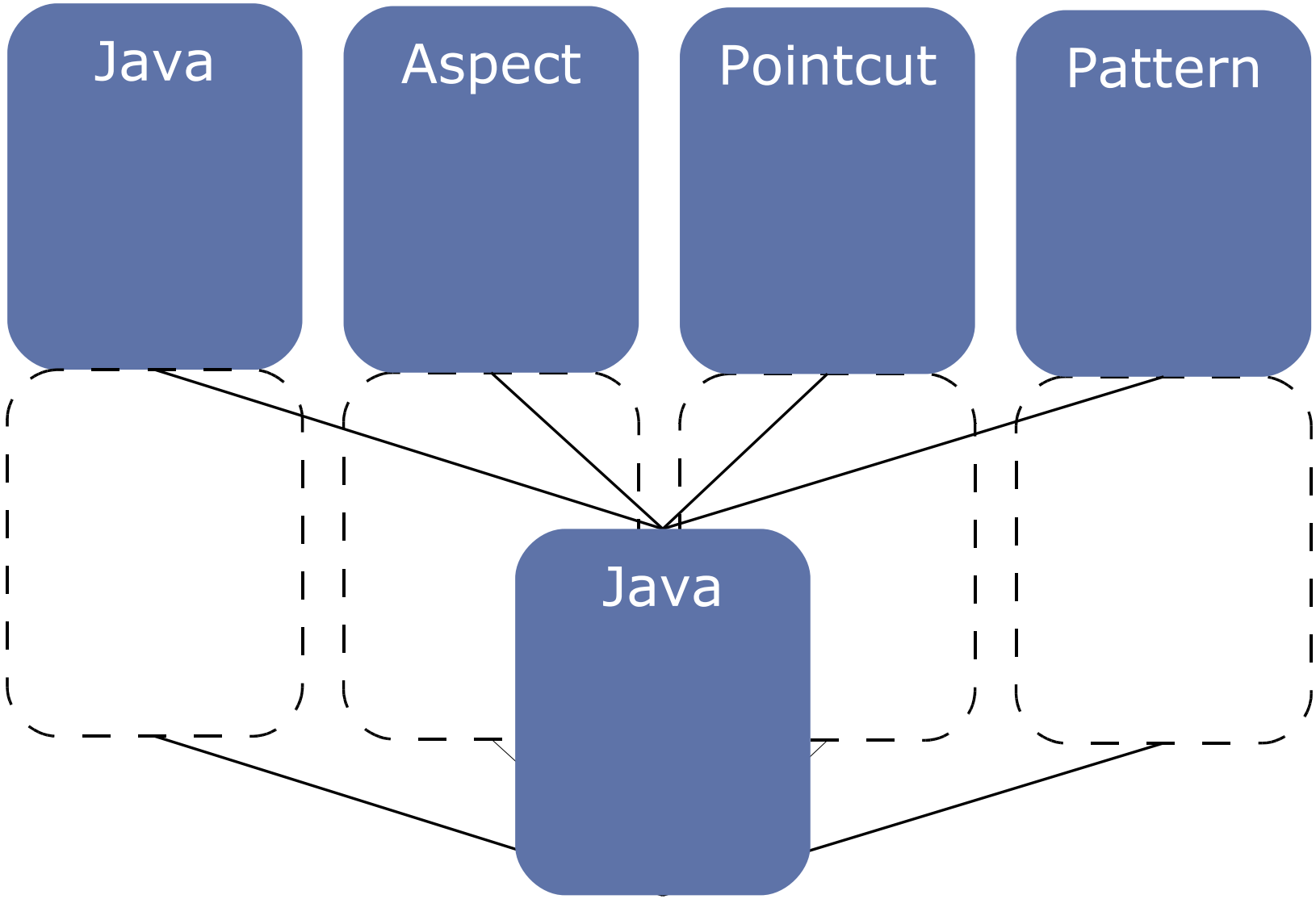

Java
Aspect
Pointcut
Pattern
Java

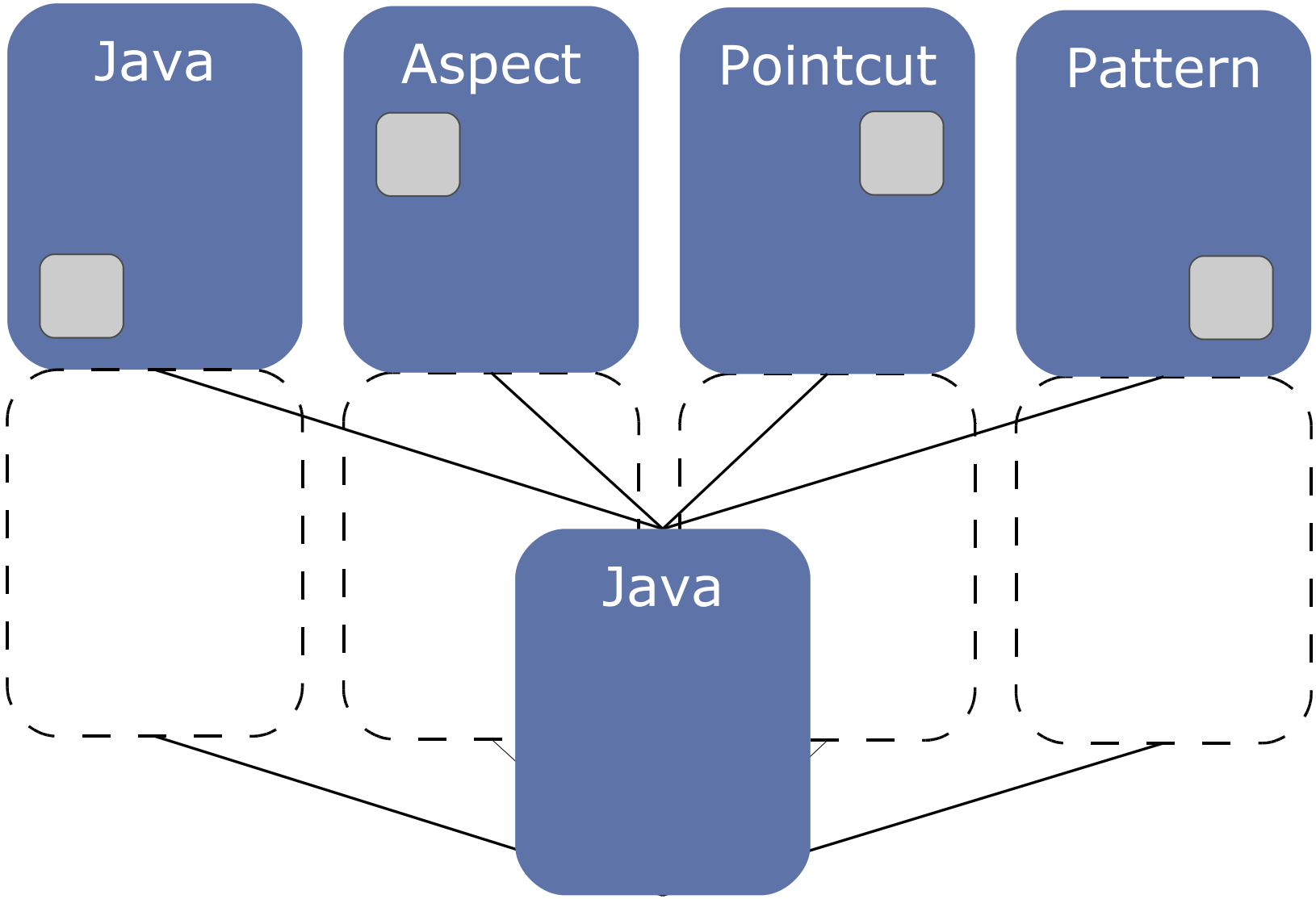

Java
Aspect
Pointcut
Pattern
Java


Java
Aspect
Pointcut
Pattern
Java


Java
Aspect
Pointcut
Pattern
Java
Java
Java
Java


Java
Aspect
Pointcut
Pattern
Java
Java
Java
Java


Java
Aspect
Pointcut
Pattern
Java


Java
Aspect
Pointcut
Pattern
Java

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec[[JavaCtx]]
ClassBodyDec[[AspectCtx]] -> AspectBodyDec
AspectDec                      -> ClassMemberDec[[JavaCtx]]
PointcutDec                    -> ClassMemberDec[[JavaCtx]]

"before" "(" {Param[[AspectCtx]] ","}* ")" -> AdvSpec
"if" "(" Expr[[JavaCtx]] ")" -> PointcutExpr
PrimType[[PatternCtx]]       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
"after" | ... | "proceed" -> Keyword[[AspectCtx]]
"error" | ... | "warning" -> Keyword[[PointcutCtx]]
PrimPointcutName           -> Keyword[[PointcutCtx]]
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec⟦JavaCtx⟧
ClassBodyDec⟦AspectCtx⟧ -> AspectBodyDec
AspectDec                      -> ClassMemberDec⟦JavaCtx⟧
PointcutDec                    -> ClassMemberDec⟦JavaCtx⟧

"before" "(" {Param⟦AspectCtx⟧ ","}* ")" -> AdvSpec
"if" "(" Expr⟦JavaCtx⟧ ")" -> PointcutExpr
PrimType⟦PatternCtx⟧       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword⟦JavaCtx⟧
"after" | ... | "proceed" -> Keyword⟦AspectCtx⟧
"error" | ... | "warning" -> Keyword⟦PointcutCtx⟧
PrimPointcutName           -> Keyword⟦PointcutCtx⟧
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec[[JavaCtx]]
ClassBodyDec[[AspectCtx]] -> AspectBodyDec
AspectDec                      -> ClassMemberDec[[JavaCtx]]
PointcutDec                    -> ClassMemberDec[[JavaCtx]]

"before" "(" {Param[[AspectCtx]] ","}* ")" -> AdvSpec
"if" "(" Expr[[JavaCtx]] ")" -> PointcutExpr
PrimType[[PatternCtx]]       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
"after" | ... | "proceed" -> Keyword[[AspectCtx]]
"error" | ... | "warning" -> Keyword[[PointcutCtx]]
PrimPointcutName           -> Keyword[[PointcutCtx]]
```

some production rules using grammar mixins

```
AspectDec                      -> TypeDec[[JavaCtx]]
ClassBodyDec[[AspectCtx]] -> AspectBodyDec
AspectDec                      -> ClassMemberDec[[JavaCtx]]
PointcutDec                    -> ClassMemberDec[[JavaCtx]]

"before" "(" {Param[[AspectCtx]] ","}* ")" -> AdvSpec
"if" "(" Expr[[JavaCtx]] ")" -> PointcutExpr
PrimType[[PatternCtx]]       -> TypePattern
```

abc's context-specific keywords

```
"privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
"after" | ... | "proceed" -> Keyword[[AspectCtx]]
"error" | ... | "warning" -> Keyword[[PointcutCtx]]
PrimPointcutName           -> Keyword[[PointcutCtx]]
```

The AspectJ syntax definition itself can be extended as well.

```
module HelloWorld[JavaCtx AspectCtx PointcutCtx]
exports context-free syntax
  "cast" "(" TypePattern ")" -> PointcutExpr

  "global" ":" ClassNamePattern ":"
                      PointcutExpr ";" -> PointcutDec

  "cflowlevel" "(" IntLiteral[[JavaCtx]] ","
                      PointcutExpr ")" -> PointcutExpr
lexical syntax
  "cast"       -> Keyword[[PointcutCtx]]
  "cflowlevel" -> Keyword[[PointcutCtx]]
  "global"     -> Keyword[[JavaCtx]]
  "global"     -> Keyword[[AspectCtx]]
```

AspectJ syntax definition

- formal
- extensible
- declarative

Enablers

- scannerless generalized-LR parsing
- modular syntax definition
- grammar mixins

Generalizing

- more languages are mixtures of various sublanguages
- challenging for specification and implementation
- strong motivation for addressing barriers to wider adaption

```
context-free syntax
  PseudoKeyword -> TypeName[[JavaCtx]]              {reject}
  PseudoKeyword -> PackageOrTypeName[[JavaCtx]]     {reject}
  PseudoKeyword -> TypeName[[AspectCtx]]            {reject}
  PseudoKeyword -> PackageOrTypeName[[AspectCtx]]   {reject}
lexical syntax
    "aspect" | "pointcut" | "privileged" | "before"
    | "after" | "around" | "declare" -> PseudoKeyword
```

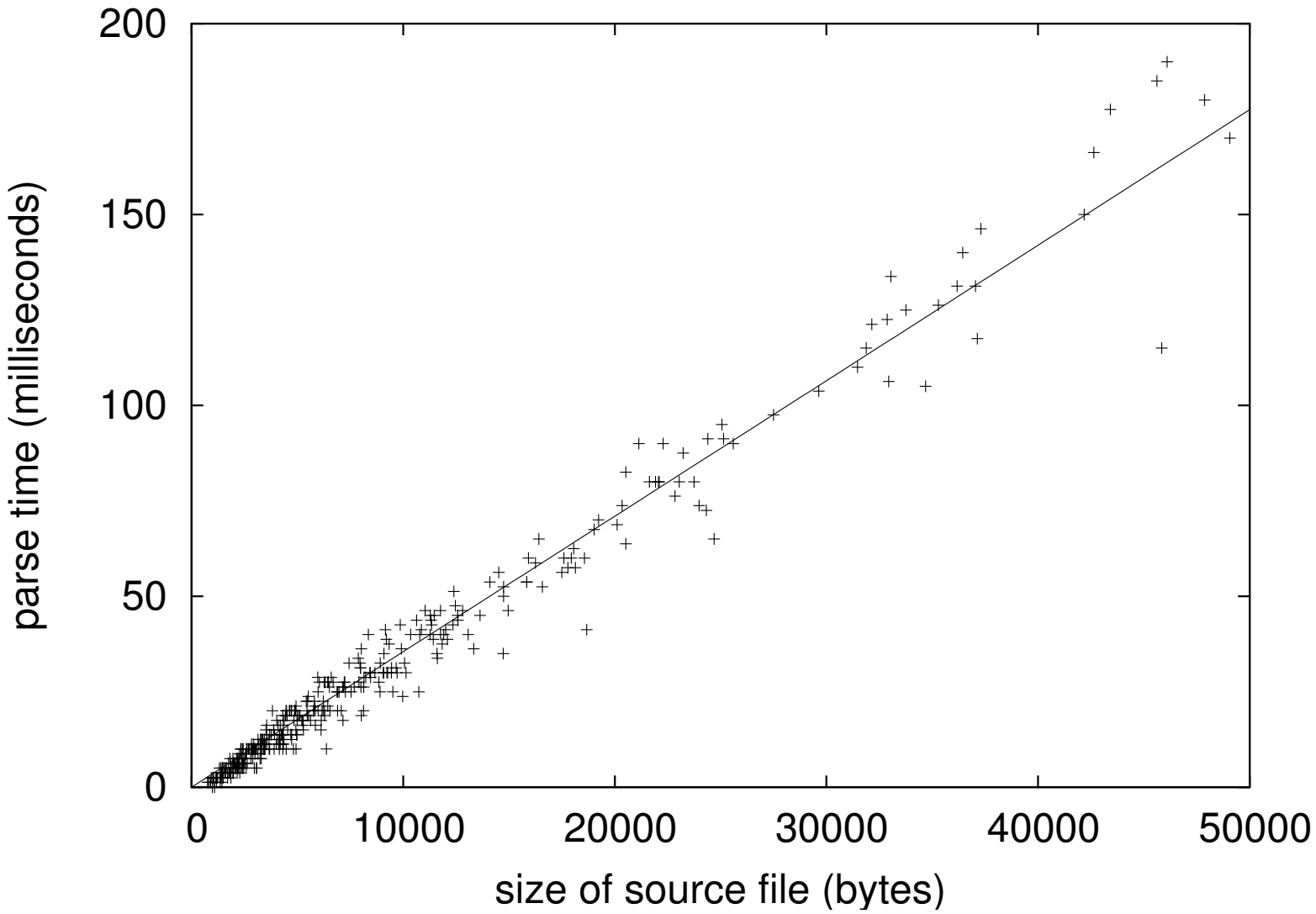
```
lexical syntax
  "privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
```

```
TypeArgs ::= '<' TypeArgList1
TypeArgList  -> TypeArg
TypeArgList  ::= TypeArgList ',' TypeArg
TypeArgList1 -> TypeArg1
TypeArgList1 ::= TypeArgList ',' TypeArg1
TypeArgList2 -> TypeArg2
TypeArgList2 ::= TypeArgList ',' TypeArg2
TypeArgList3 -> TypeArg3
TypeArgList3 ::= TypeArgList ',' TypeArg3
TypeArg  ::= RefType
TypeArg1 -> RefType1
TypeArg2 -> RefType2
TypeArg3 -> RefType3
RefType1 ::= RefType '>'
RefType1 ::= ClassOrInterface '<' TypeArgList2
RefType2 ::= RefType '>>'
RefType2 ::= ClassOrInterface '<' TypeArgList3
RefType3 ::= RefType '>>>'
```

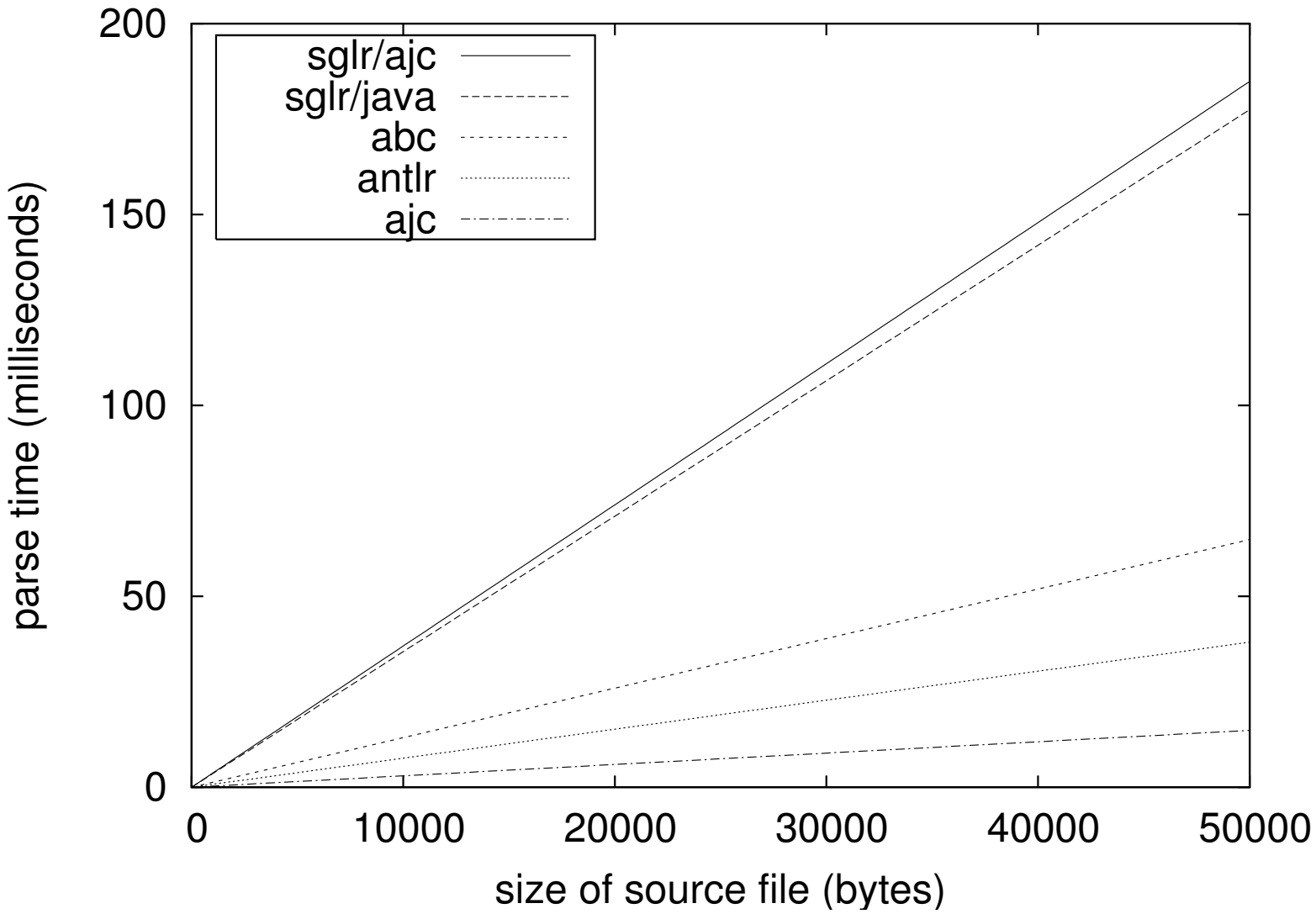
```
module JavaMix[Ctx]
imports Java
  [ CompilationUnit => CompilationUnit⟦Ctx⟧
    TypeDec         => TypeDec⟦Ctx⟧
    ...
    FieldAccess     => FieldAccess⟦Ctx⟧
    MethodSpec      => MethodSpec⟦Ctx⟧
    Expr            => Expr⟦Ctx⟧  ]
```

```
module AspectJ[JavaCtx AspectCtx PointcutCtx PatternCtx]
imports
  JavaMix[JavaCtx]
  JavaMix[AspectCtx]
  JavaMix[PointcutCtx]
  JavaMix[PatternCtx]
  aspect/Declaration[AspectCtx JavaCtx]
  pattern/Main[PatternCtx]
  pointcut/Expression[PointcutCtx JavaCtx]
```

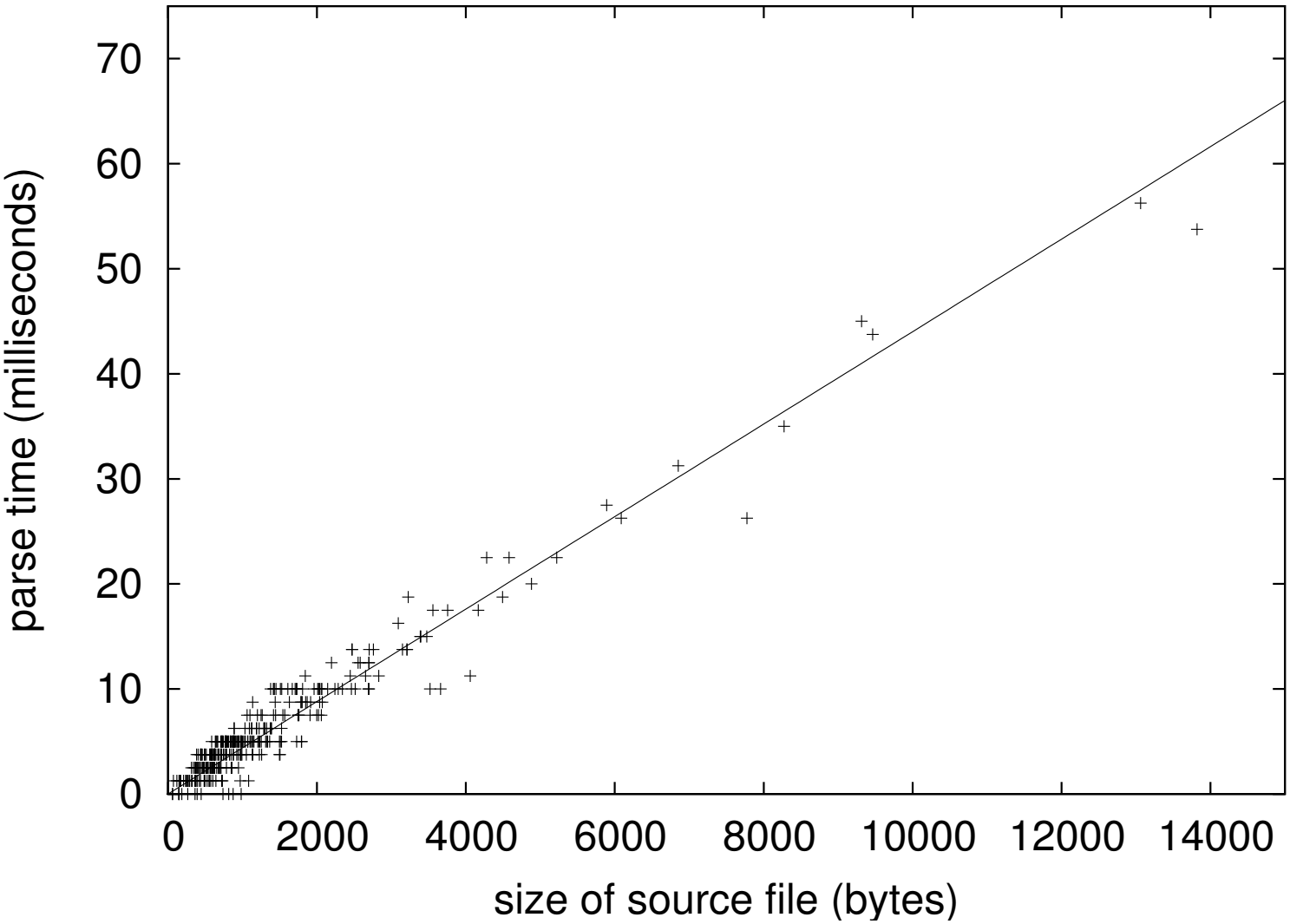
Performance: Java Benchmark



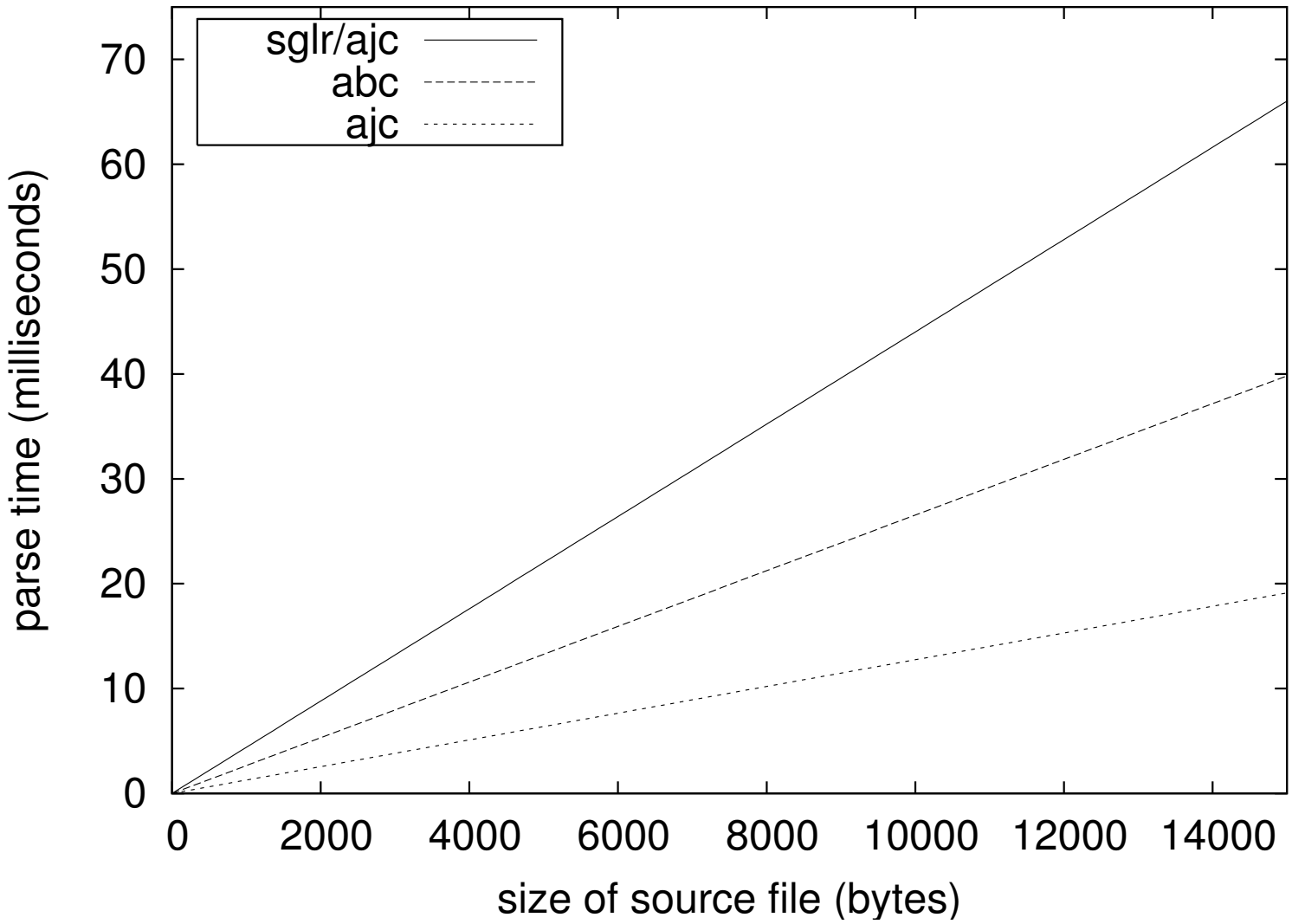
Performance: Java Benchmark



Performance: AspectJ Benchmark



Performance: AspectJ Benchmark



```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

PseudoTokens ::=
  one or more PseudoToken or ':'
PseudoTokensNoColon ::=
  one or more PseudoToken
```

```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

PseudoTokens ::=
  one or more PseudoToken or ':'
PseudoTokensNoColon ::=
  one or more PseudoToken
```

```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

PseudoTokens ::=
  one or more PseudoToken or ':'
PseudoTokensNoColon ::=
  one or more PseudoToken
```

```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

PseudoTokens ::=
  one or more PseudoToken or ':'
PseudoTokensNoColon ::=
  one or more PseudoToken
```

Enter PointcutIf state: keyword lexer action

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

Leave PointcutIf state: parentheses matching

```
<POINTCUTIFEXPR> {
  "(" { parenLevel++; return op(sym.LPAREN); }
  ")" { parenLevel--;
        if(parenLevel == nestingStack.peek().parenLevel)
          returnToPrevState();
        return op(sym.RPAREN);
      }
}
```

Enter PointcutIf state: keyword lexer action

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

Leave PointcutIf state: parentheses matching

```
<POINTCUTIFEXPR> {
  "(" { parenLevel++; return op(sym.LPAREN); }
  ")" { parenLevel--;
        if(parenLevel == nestingStack.peek().parenLevel)
          returnToPrevState();
        return op(sym.RPAREN);
      }
}
```

Enter PointcutIf state: keyword lexer action

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

Leave PointcutIf state: parentheses matching

```
<POINTCUTIFEXPR> {
  "(" { parenLevel++; return op(sym.LPAREN); }
  ")" { parenLevel--;
        if(parenLevel == nestingStack.peek().parenLevel)
          returnToPrevState();
        return op(sym.RPAREN);
      }
}
```

Enter PointcutIf state: keyword lexer action

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

Leave PointcutIf state: parentheses matching

```
<POINTCUTIFEXPR> {
  "(" { parenLevel++; return op(sym.LPAREN); }
  ")" { parenLevel--;
        if(parenLevel == nestingStack.peek().parenLevel)
          returnToPrevState();
        return op(sym.RPAREN);
      }
}
```

Scanners reserve keywords

- Solution: explicitly allow keywords

```
JavaIdentifier -> 'Identifier'
JavaIdentifier -> AjSimpleName
AjSimpleName -> 'around' or 'aspect' or 'privileged' or
    'pointcut' or 'before' or 'after' or 'declare'
```

- JavaIdentifier used where possible

```
ClassHeaderName1 ::= Modifiersopt 'class' JavaIdentifier
MethodHeaderName ::= Modifiersopt Type JavaIdentifier '('
```

- For LALR reasons not as simple typename

```
import privileged.*;
import org.privileged.*;
```