COMP 520 - Compilers

Lecture 18 – Abstract Syntax Trees, Why??

Reminders

- Midterm 2 on Thursday, 4/11
- Make sure you have some experience in PA4

Announcements

- I have finished reading all PA3 submissions done by Sunday night (keep working if you’re not done).
- This lecture is targeted towards making sure we all understand the power of ASTs.

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Announcements (2)

- A small portion of you maximized AST potential.
- A good portion nearly maximized their potential, and a good portion had a lot of redundant code.
- Today we uncover the secrets of ASTs.

Why are we using ASTs?

- Is it just some arbitrary decision that is just "one way to do Compilers"?

Why are we using ASTs?

- Is it just some arbitrary decision that is just "one way to do Compilers"?
- Yes and no. Yes it is, but it makes life easier for us. No, it will make life harder for you otherwise, so we should learn ASTs.
    - *Examples of "no": bogo sort*
- (Note: with some language exceptions, ASTs can look vastly different, but the core concept remains the same).
- Today's question: did ASTs make our lives easier?

Visitor Model

A quick review

AST

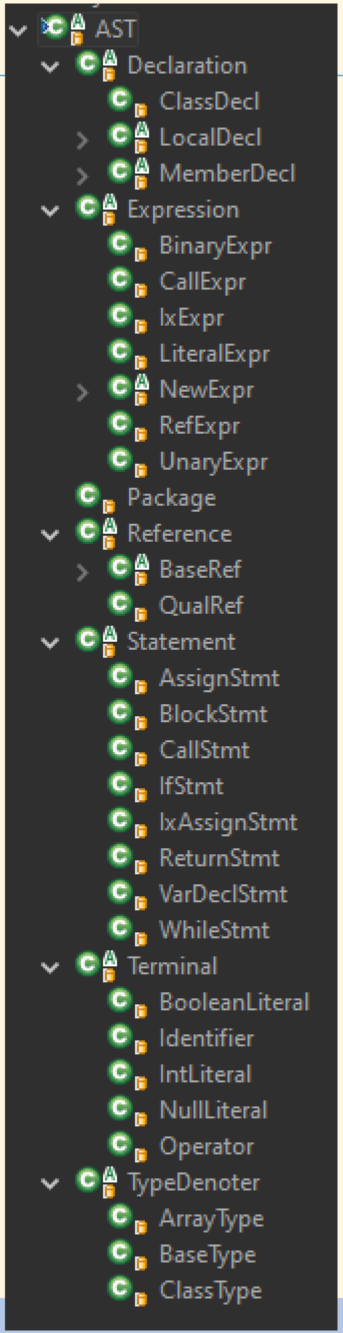
```
public abstract class AST {
    public abstract <A,R> R visit(Visitor<A,R> v, A o);
}
```

This is an abstract class.
This method is not defined.

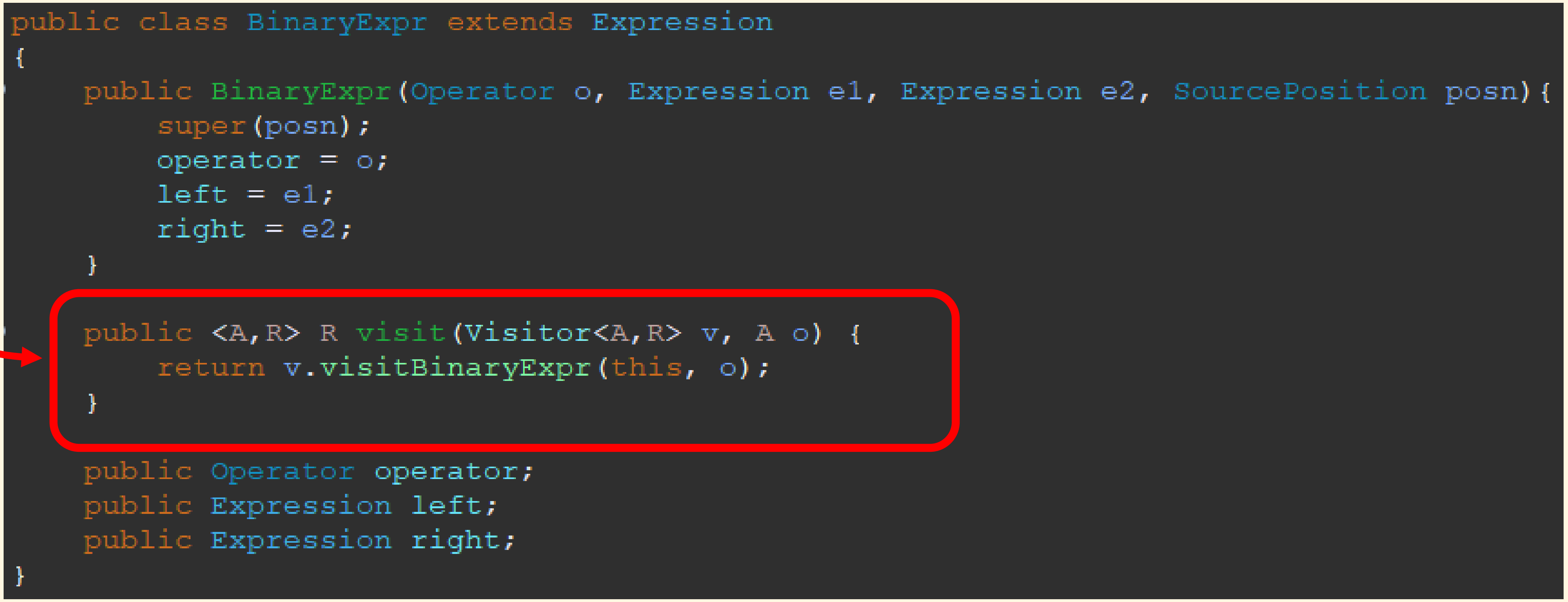
AST (2)

```
public abstract class AST {
    public abstract <A,R> R visit(Visitor<A,R> v, A o);
}
```

Visit is defined in concrete classes:


```
AST
  Declaration
    ClassDecl
    LocalDecl
    MemberDecl
  Expression
    BinaryExpr
    CallExpr
    IxExpr
    LiteralExpr
    NewExpr
    RefExpr
    UnaryExpr
  Package
  Reference
    BaseRef
    QualRef
  Statement
    AssignStmt
    BlockStmt
    CallStmt
    IfStmt
    IxAssignStmt
    ReturnStmt
    VarDeclStmt
    WhileStmt
  Terminal
    BooleanLiteral
    Identifier
    IntLiteral
    NullLiteral
    Operator
  TypeDenoter
    ArrayType
    BaseType
    ClassType
```

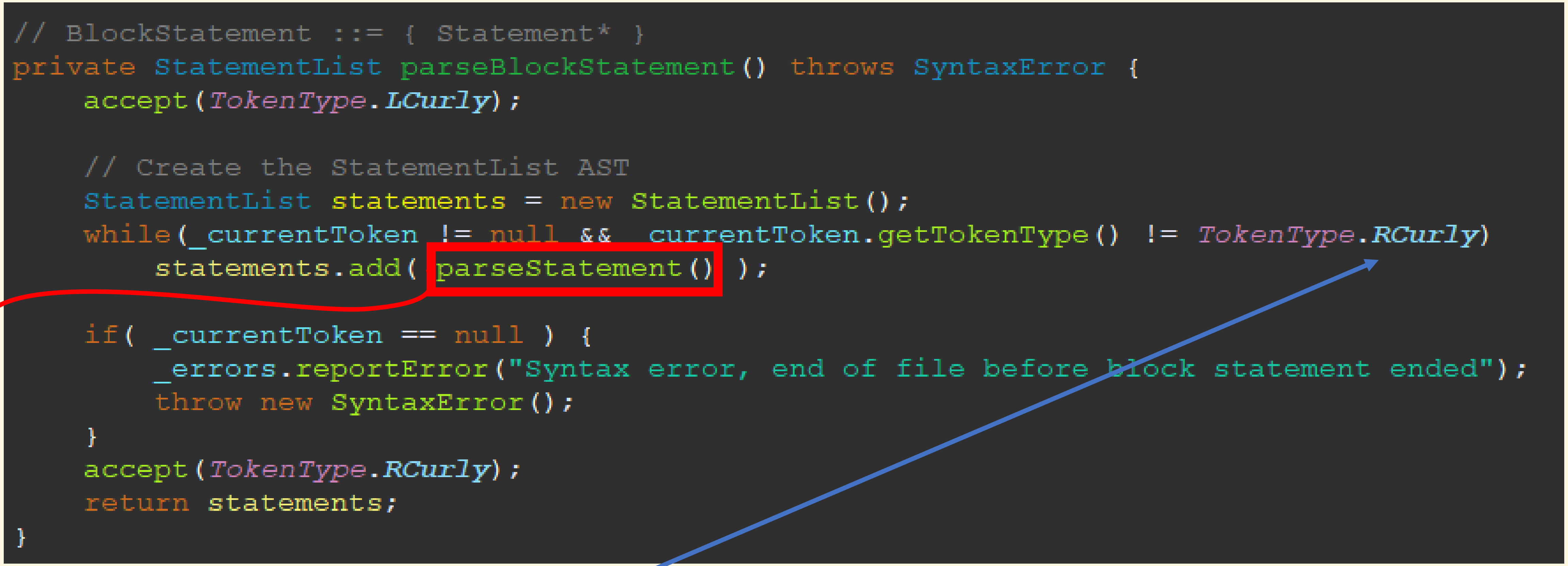
Concrete Visit Method



```
public class BinaryExpr extends Expression
{
    public BinaryExpr(Operator o, Expression e1, Expression e2, SourcePosition posn){
        super(posn);
        operator = o;
        left = e1;
        right = e2;
    }

    public <A,R> R visit(Visitor<A,R> v, A o) {
        return v.visitBinaryExpr(this, o);
    }

    public Operator operator;
    public Expression left;
    public Expression right;
}
```

Let’s build the framework to see this in action.

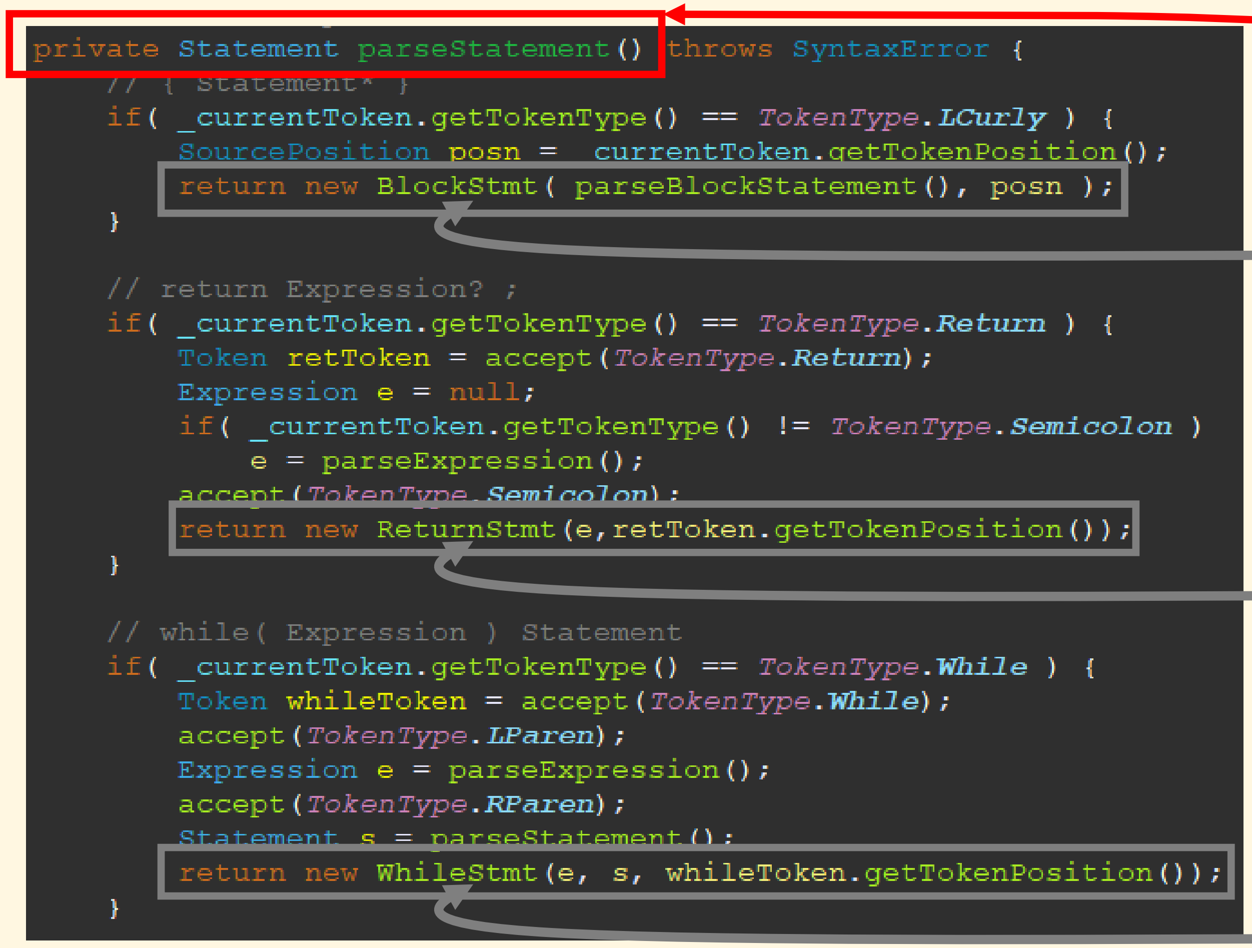
Recall Creating Statement Lists



```
// BlockStatement ::= { Statement* }
private StatementList parseBlockStatement() throws SyntaxError {
    accept(TokenType.LCurly);

    // Create the StatementList AST
    StatementList statements = new StatementList();
    while(_currentToken != null &&  currentToken.getTokenType() != TokenType.RCurly)
        statements.add( parseStatement() );

    if( _currentToken == null ) {
        _errors.reportError("Syntax error, end of file before block statement ended");
        throw new SyntaxError();
    }
    accept(TokenType.RCurly);
    return statements;
}
```

"So long as input isn't a }, keep giving me an abstract `Statement` object returned by `parseStatement();`"

A peek into parseStatement



```
private Statement parseStatement() throws SyntaxError {
    // { Statement* }
    if( _currentToken.getTokenType() == TokenType.LCurly ) {
        SourcePosition posn = _currentToken.getTokenPosition();
        return new BlockStmt( parseBlockStatement(), posn );
    }

    // return Expression? ;
    if( _currentToken.getTokenType() == TokenType.Return ) {
        Token retToken = accept(TokenType.Return);
        Expression e = null;
        if( _currentToken.getTokenType() != TokenType.Semicolon )
            e = parseExpression();
        accept(TokenType.Semicolon);
        return new ReturnStmt(e,retToken.getTokenPosition());
    }

    // while( Expression ) Statement
    if( _currentToken.getTokenType() == TokenType.While ) {
        Token whileToken = accept(TokenType.While);
        accept(TokenType.LParen);
        Expression e = parseExpression();
        accept(TokenType.RParen);
        Statement s = parseStatement();
        return new WhileStmt(e, s, whileToken.getTokenPosition());
    }
```

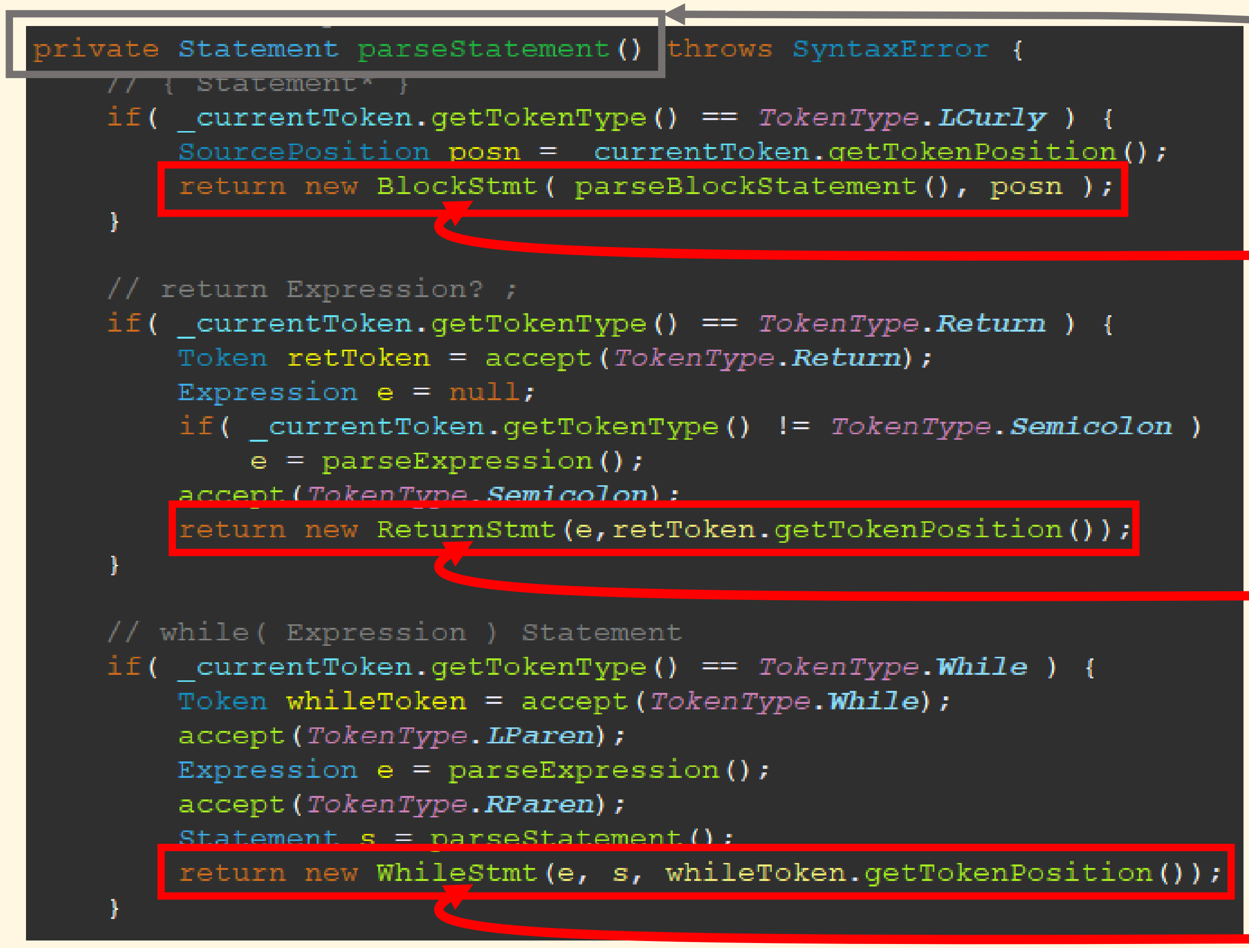
"This method will return a Statement. What kind of statement? No idea!"

BlockStmt, which extends Statement and has a defined visit method.

ReturnStmt, which extends Statement and has a defined visit method.

WhileStmt, which extends Statement and has a defined visit method.

A peek into visitStatement (2)



```
private Statement parseStatement() throws SyntaxError {
    // { Statement* }
    if( _currentToken.getTokenType() == TokenType.LCurly ) {
        SourcePosition posn = _currentToken.getTokenPosition();
        return new BlockStmt( parseBlockStatement(), posn );
    }

    // return Expression? ;
    if( _currentToken.getTokenType() == TokenType.Return ) {
        Token retToken = accept(TokenType.Return);
        Expression e = null;
        if( _currentToken.getTokenType() != TokenType.Semicolon )
            e = parseExpression();
        accept(TokenType.Semicolon);
        return new ReturnStmt(e,retToken.getTokenPosition());
    }

    // while( Expression ) Statement
    if( _currentToken.getTokenType() == TokenType.While ) {
        Token whileToken = accept(TokenType.While);
        accept(TokenType.LParen);
        Expression e = parseExpression();
        accept(TokenType.RParen);
        Statement s = parseStatement();
        return new WhileStmt(e, s, whileToken.getTokenPosition());
    }
```

"This method will return a Statement. What kind of statement? No idea!"

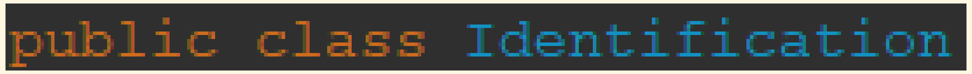
BlockStmt, which extends Statement and has a defined visit method.

ReturnStmt, which extends Statement and has a defined visit method.

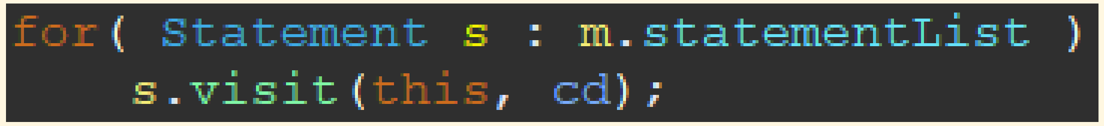
WhileStmt, which extends Statement and has a defined visit method.

Calling an Abstract Method

ME=Identification



```
public class Identification
```

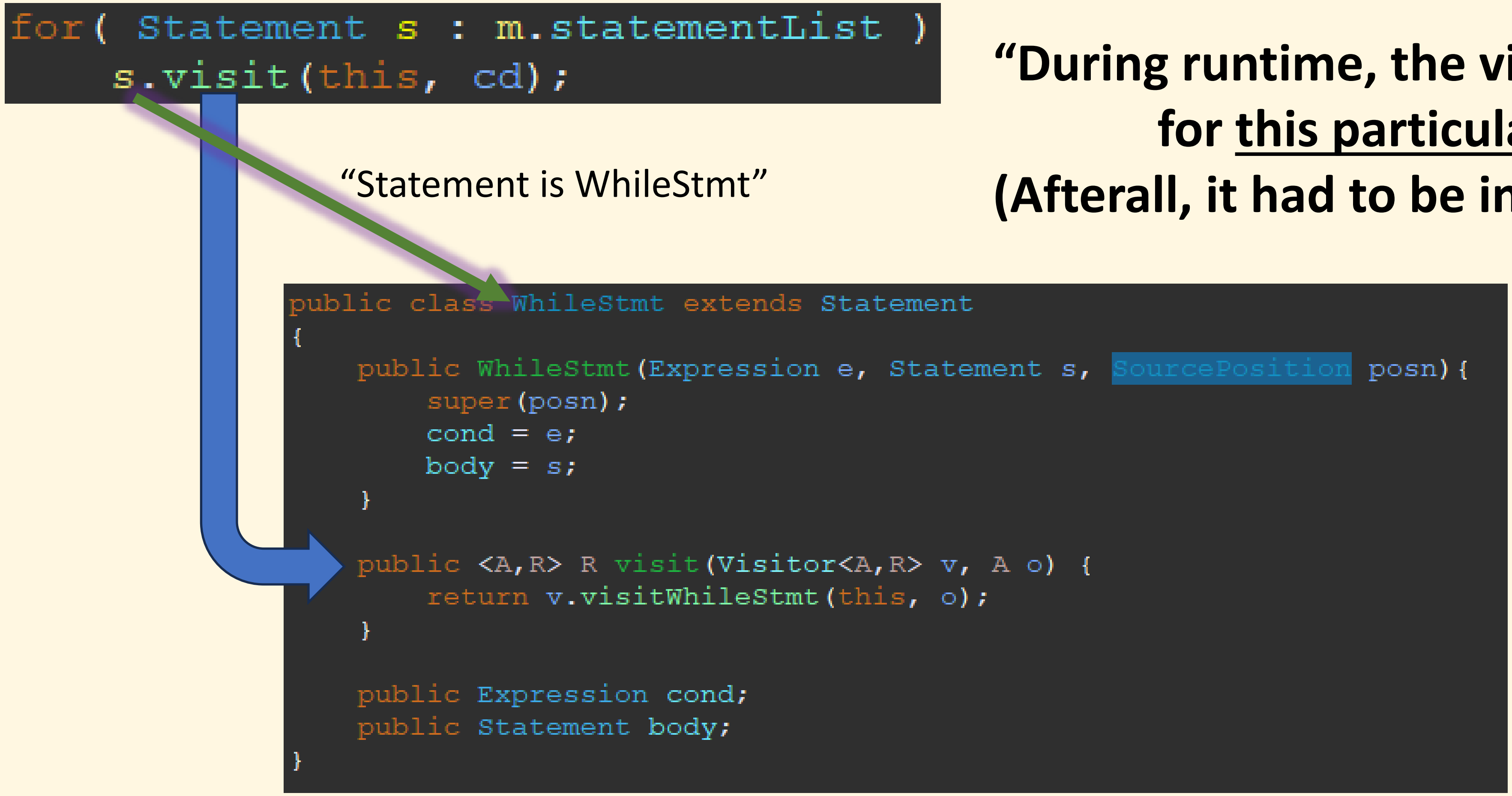


```
for( Statement s : m.statementList )
    s.visit(this, cd);
```

“I don’t know what type this Statement “s” is, but I know it was instantiated with a REAL visit method”

Why?

Calling an Abstract Method (2)



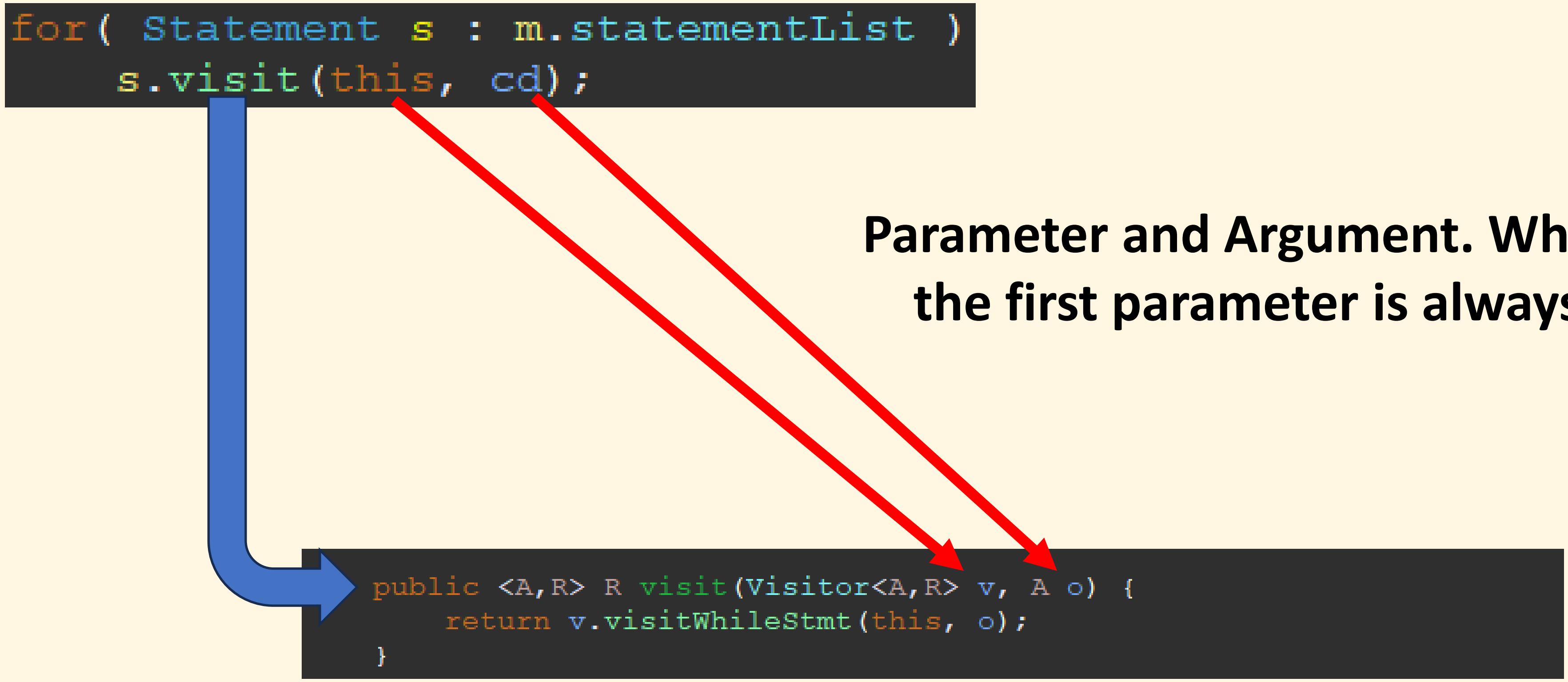
```
for( Statement s : m.statementList )
    s.visit(this, cd);
```

```
public class WhileStmt extends Statement
{
    public WhileStmt(Expression e, Statement s, SourcePosition posn){
        super(posn);
        cond = e;
        body = s;
    }

    public <A,R> R visit(Visitor<A,R> v, A o) {
        return v.visitWhileStmt(this, o);
    }

    public Expression cond;
    public Statement body;
}
```

“During runtime, the visit method maps here for <u>this particular</u> example Statement”

(Afterall, it had to be instantiated somehow)

Calling an Abstract Method (3)

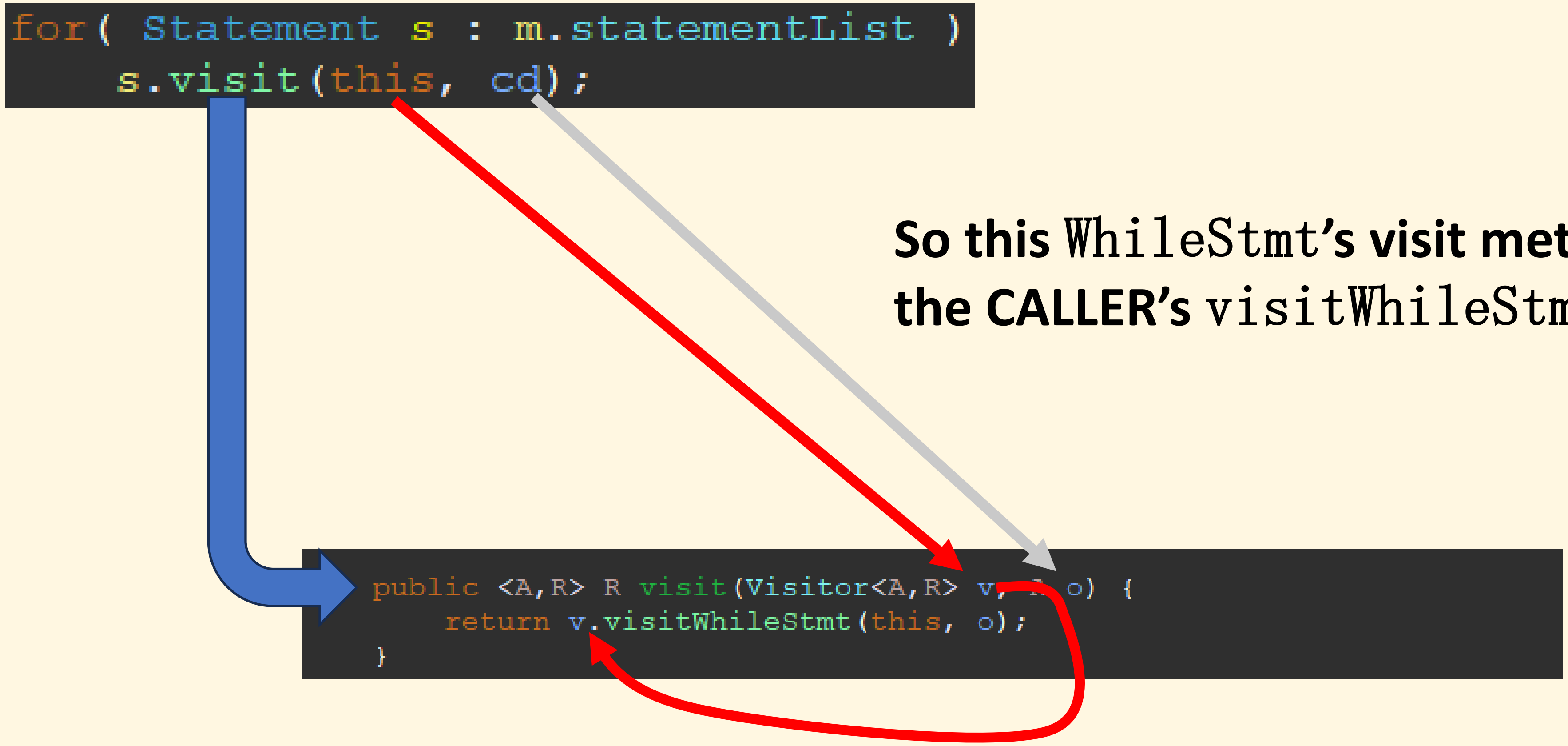


```
for( Statement s : m.statementList )
    s.visit(this, cd);
```

```
public <A,R> R visit(Visitor<A,R> v, A o) {
    return v.visitWhileStmt(this, o);
}
```

Parameter and Argument. When visiting, the first parameter is always "this"

Calling an Abstract Method (4)



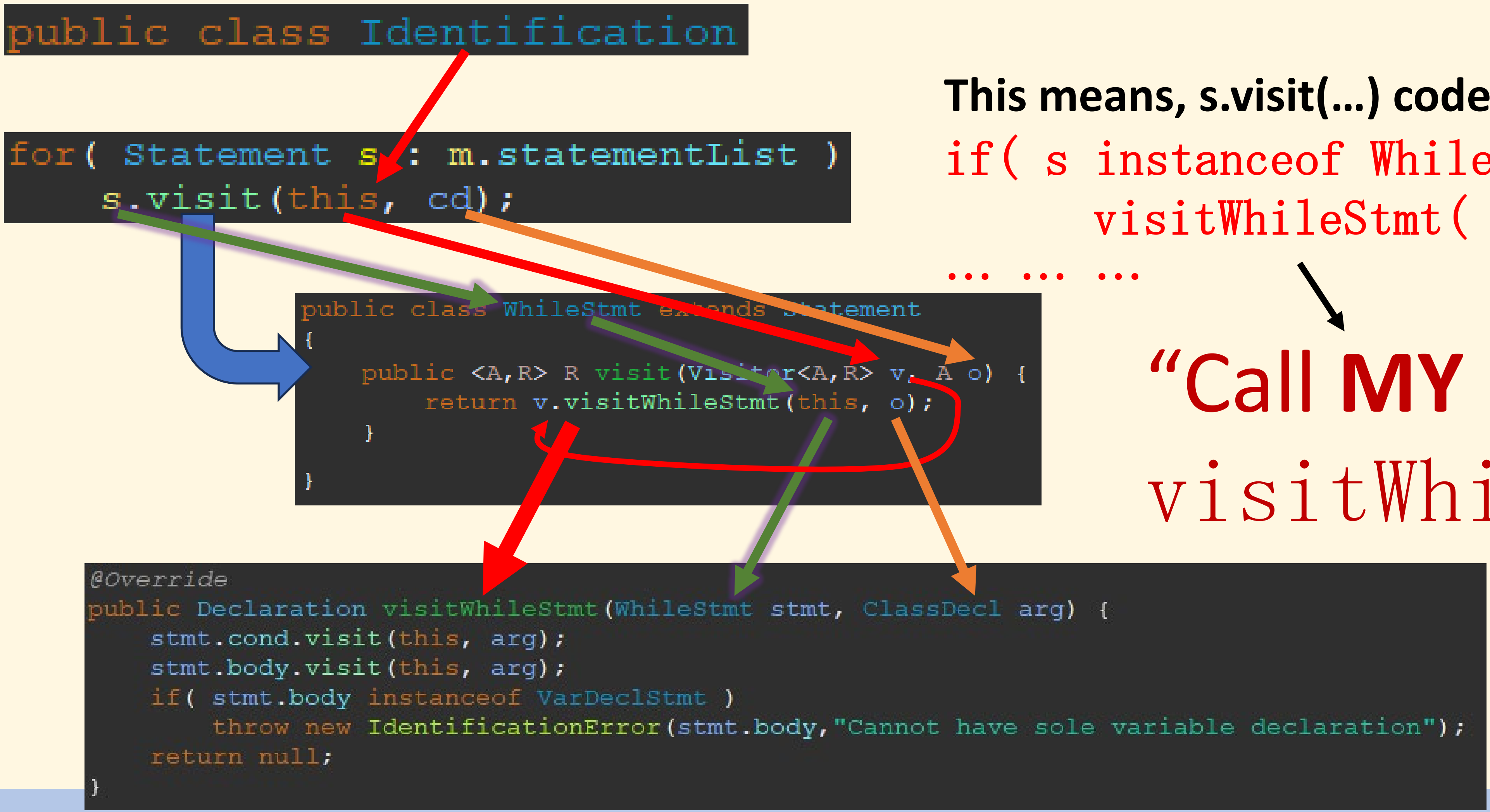
```
for( Statement s : m.statementList )
    s.visit(this, cd);
```

```
public <A,R> R visit(Visitor<A,R> v, A o) {
    return v.visitWhileStmt(this, o);
}
```

So this `WhileStmt`'s visit method just calls the CALLER's `visitWhileStmt` method.

Calling an Abstract Method (5)

ME



```
public class Identification
```

```
for( Statement s : m.statementList )
    s.visit(this, cd);
```

```
public class WhileStmt extends Statement
{
    public <A,R> R visit(Visitor<A,R> v, A o) {
        return v.visitWhileStmt(this, o);
    }

}
```

```
@Override
public Declaration visitWhileStmt(WhileStmt stmt, ClassDecl arg) {
    stmt.cond.visit(this, arg);
    stmt.body.visit(this, arg);
    if( stmt.body instanceof VarDeclStmt )
        throw new IdentificationError(stmt.body,"Cannot have sole variable declaration");
    return null;
}
```

This means, s.visit(…) code is equivalent to:

```
if( s instanceof WhileStmt )
        visitWhileStmt( s, arg );
… … …
```

“Call **MY** visitWhileStmt”

PA2 - ASTs

- In case you forgot, **you** instantiated the concrete classes in PA2 in your parser.

- So when you instantiated a concrete “`VardeclStmt`”, you specified “if you visit this `Statement`, make sure you pair it with ‘`visitVarDeclStmt`’ for whoever the visitor is.”

The Terrifying, Terrific, Tantalizing, Tormenting Truth of ASTs

Elegantly combine parsing and input code traversal.

Identification

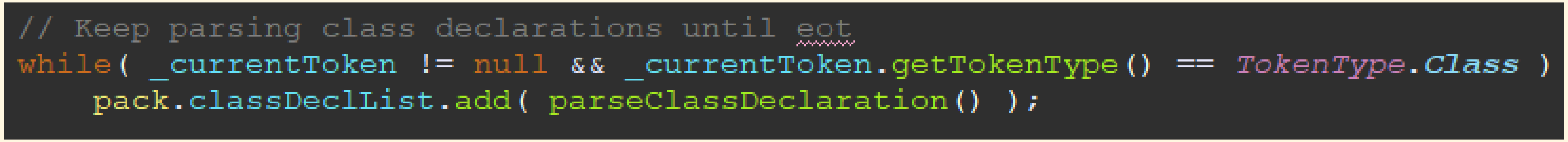
- Consider wanting to implement this:

```
visitIdentifier( Identifier id, String ctx ) ::=
    id.decl = findDeclaration( ctx, id );
    return id.decl;
```

- (It won’t know the context, needs it to be passed in)

Package…

- Package contains: ClassDeclList, which is just a list of ClassDecls
- Parser: parseClassDecl- Returns a ClassDecl.



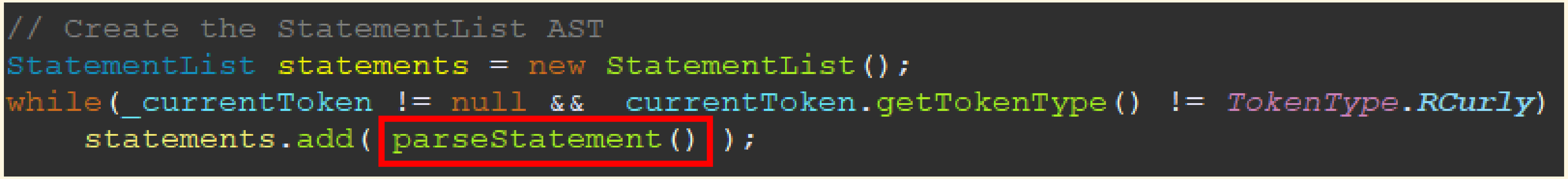
```
// Keep parsing class declarations until eot
while( _currentToken != null && _currentToken.getTokenType() == TokenType.Class )
    pack.classDeclList.add( parseClassDeclaration() );
```

- Add each ClassDecl into our list

ClassDecl

- Contains a FieldDeclList (member variables) and MethodDeclList (member methods)

- If it was a method, you would `parseStatement` until out of statements, and store that in the `MethodDecl`.



```
// Create the StatementList AST
StatementList statements = new StatementList();
while(_currentToken != null &&  currentToken.getTokenType() != TokenType.RCurly)
    statements.add( parseStatement() );
```

Visiting a Method

- Thus, when you visit the `Statement`s in a `MethodDecl`, you visit the `Statement` objects that you instantiated in PA2.

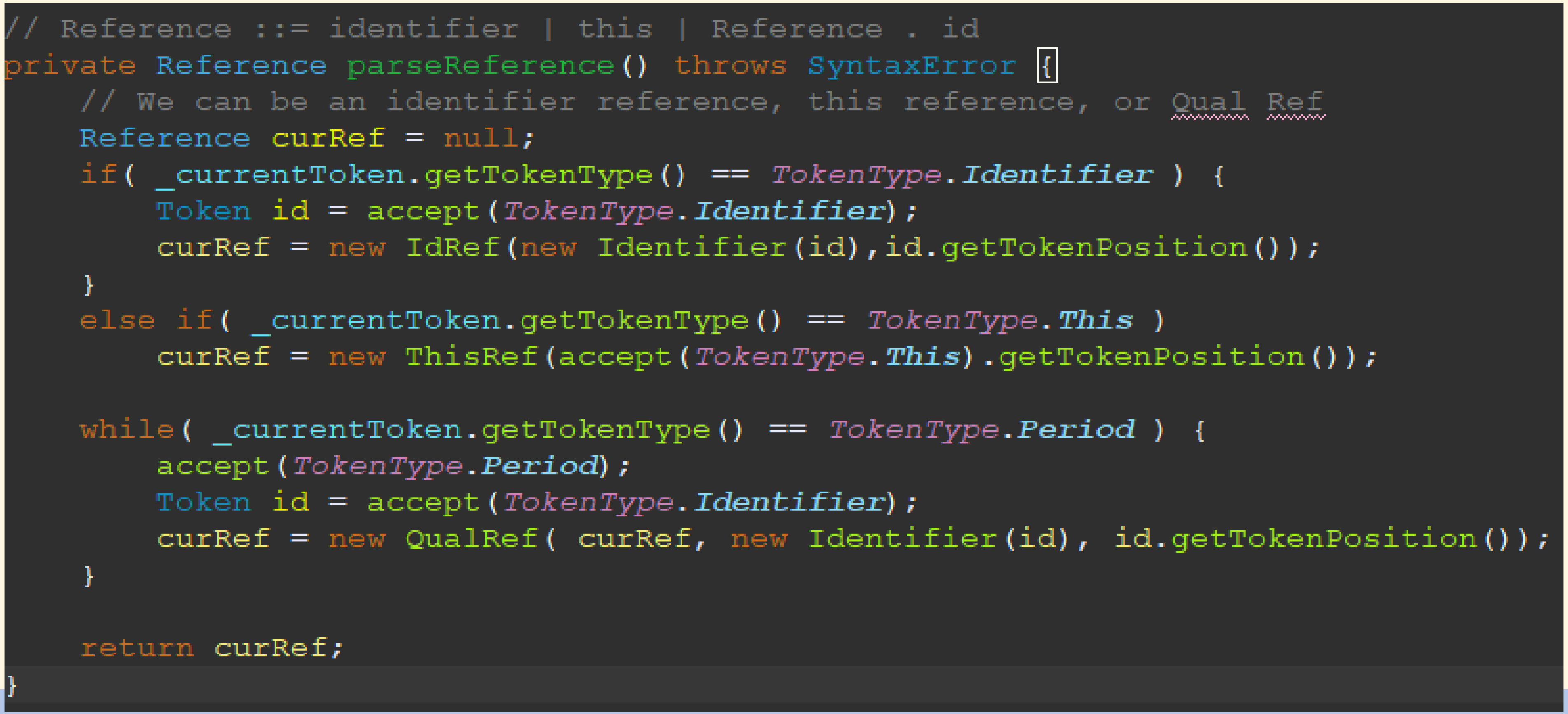
- Those statements contained `Expression`s that you also created in PA2.

The Mysteries of References

Recall: Creating References

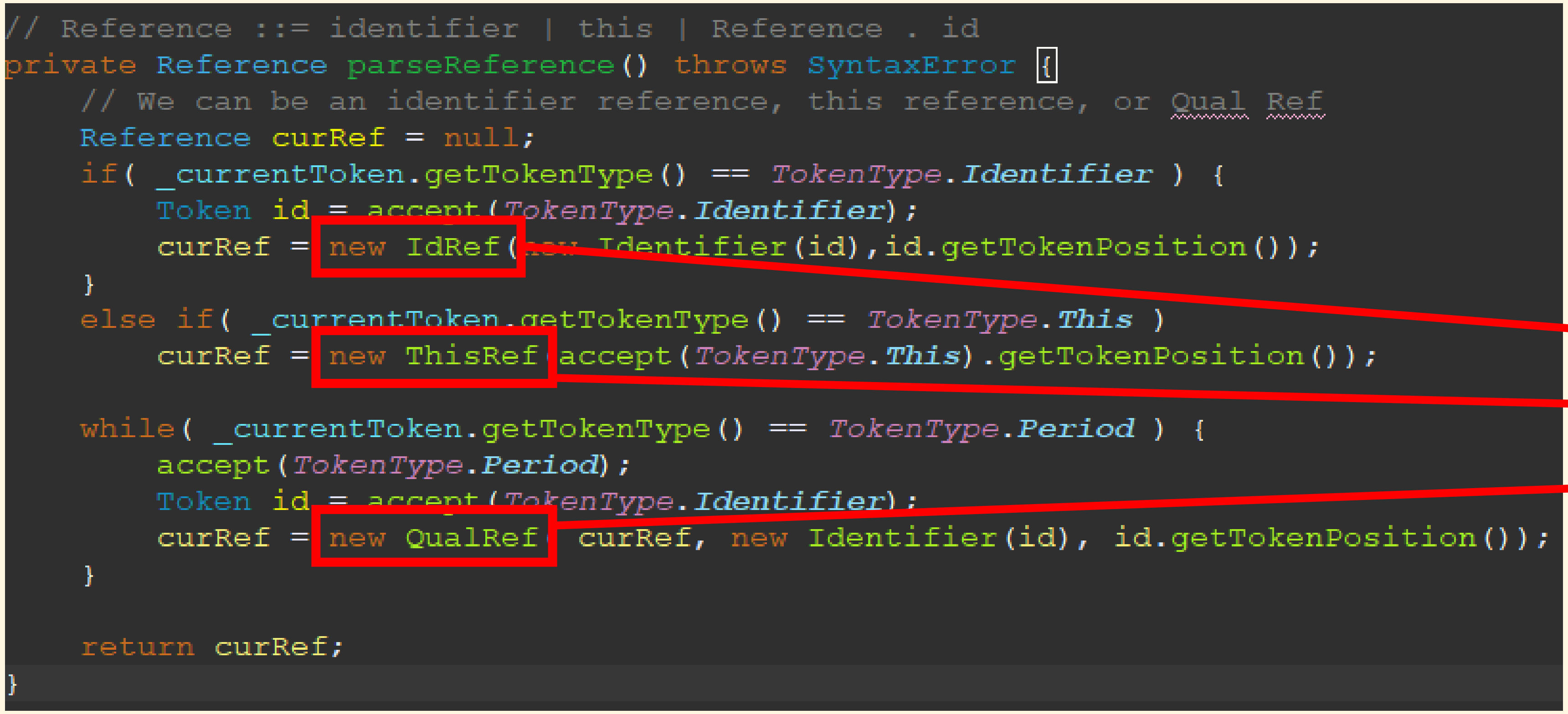
Grammar: Reference ::= id | this | Reference . id

Improved: (this|id)(.id)*



```
// Reference ::= identifier | this | Reference . id
private Reference parseReference() throws SyntaxError {
    // We can be an identifier reference, this reference, or Qual Ref
    Reference curRef = null;
    if( _currentToken.getTokenType() == TokenType.Identifier ) {
        Token id = accept(TokenType.Identifier);
        curRef = new IdRef(new Identifier(id),id.getTokenPosition());
    }
    else if( _currentToken.getTokenType() == TokenType.This )
        curRef = new ThisRef(accept(TokenType.This).getTokenPosition());

    while( _currentToken.getTokenType() == TokenType.Period ) {
        accept(TokenType.Period);
        Token id = accept(TokenType.Identifier);
        curRef = new QualRef( curRef, new Identifier(id), id.getTokenPosition());
    }

    return curRef;
}
```

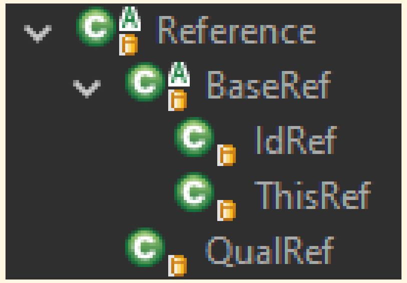
Recall: Creating References (2)



```
// Reference ::= identifier | this | Reference . id
private Reference parseReference() throws SyntaxError {
    // We can be an identifier reference, this reference, or Qual Ref
    Reference curRef = null;
    if( _currentToken.getTokenType() == TokenType.Identifier ) {
        Token id = accept(TokenType.Identifier);
        curRef = new IdRef(new Identifier(id),id.getTokenPosition());
    }
    else if( _currentToken.getTokenType() == TokenType.This )
        curRef = new ThisRef(accept(TokenType.This).getTokenPosition());

    while( _currentToken.getTokenType() == TokenType.Period ) {
        accept(TokenType.Period);
        Token id = accept(TokenType.Identifier);
        curRef = new QualRef( curRef, new Identifier(id), id.getTokenPosition());
    }

    return curRef;
}
```

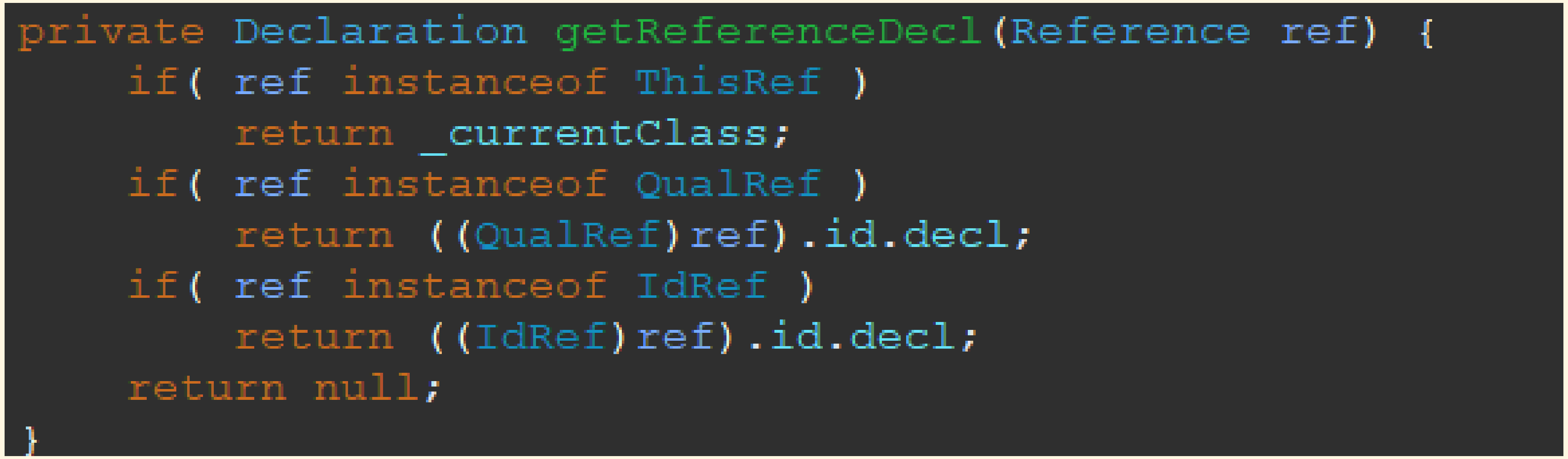
Implies: only three types of references.

IdRef,
ThisRef,
QualRef.



Confirmed

Note: If Identifiers have a Decl, then References map to a Decl too!



```
private Declaration getReferenceDecl(Reference ref) {
    if( ref instanceof ThisRef )
        return _currentClass;
    if( ref instanceof QualRef )
        return ((QualRef)ref).id.decl;
    if( ref instanceof IdRef )
        return ((IdRef)ref).id.decl;
    return null;
}
```

Note: If Identifiers have a Decl, then References map to a Decl too!

- Thus, references map to some memory location. See CFGs:
  - Reference [Expression] - Reference because it must be a variable
  - Reference (ArgumentList?) - Reference because it must be a method
  - Reference = Expression ; - Reference because need to store data.

- Makes sense, use references to “refer to something” whereas expressions need to be evaluated.

Parsing VS Grammar

- Grammar:
  - Reference ::= id | this | Reference . id
- Parsing:
  - Reference ::= (this|id)(.id)*

- **Parsing is misleading!**

Parsing VS Grammar (2)

- Grammar:
  - Original: Reference ::= id | this | Reference . id
  - Try this: **QualRef ::= (ThisRef|IdRef)(.id)+**
  - Useful to view them as AST grammars instead!

- **Question: What is the only repeating component?**

Parsing VS Grammar (3)

- Grammar:
  - Reference ::= id | this | Reference . id
  - Try this: QualRef ::= (ThisRef|IdRef)(.id)+
  - The only repeating element is **identifiers**.

- AST Example:

  IdRef . Identifier . Identifier . Identifier

IdRef

- Thus, if you visit an IdRef, Then it was either a plain “id” used as a reference, or the **left-most** identifier in a QualRef.
- THEREFORE: visitIdRef is **ALWAYS** in the context of the current class.

QualRef	Left-Hand-Side (ref)	Right-Hand-Side (id)
a.b.c.x	a.b.c (QualRef) **VISIT**	x
a.b.c	a.b (QualRef) **VISIT**	c
a.b	a (IdRef) **VISIT**	b

IdRef (2)

- Thus, if you visit an IdRef, Then it was either a plain "id" used as a reference, or the left-most identifier in a QualRef.
- THEREFORE: visitIdRef is **ALWAYS** in the context of the current class.

- And the implementation for PA3 is…

```
visitIdRef ::=
    ref.id.visit( this, currentClassName );
    // (CTX is currentClass)
```

THE UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

IdRef (Recall visitIdentifier)

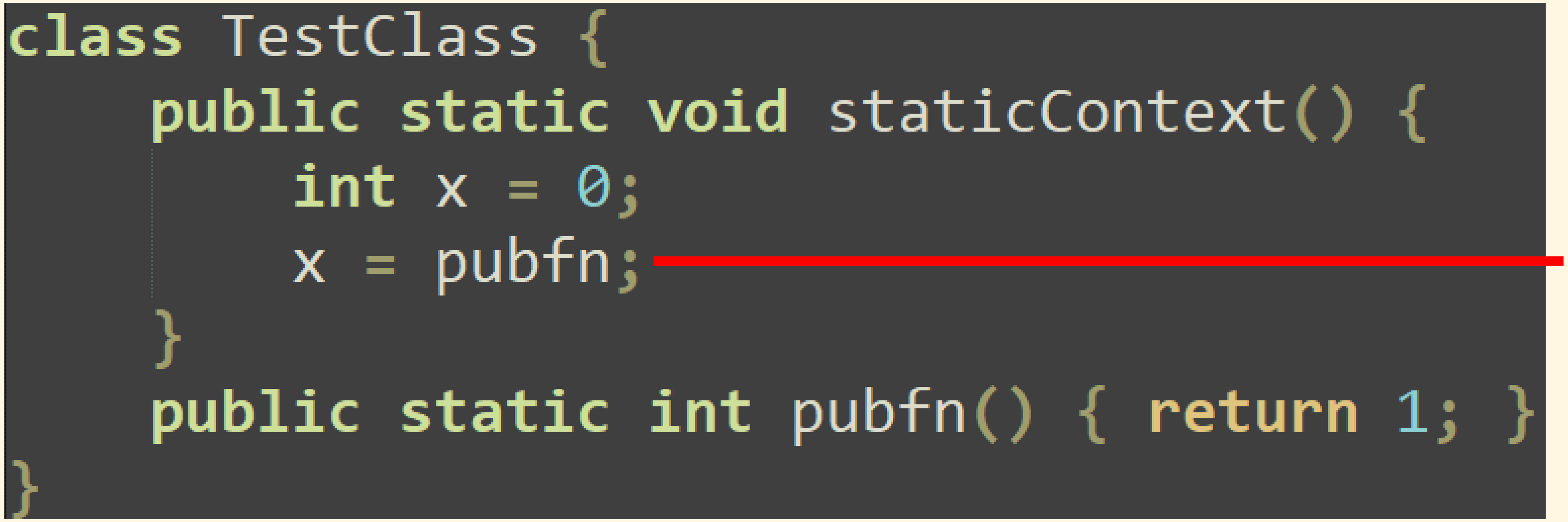
```
visitIdRef ::=
    ref.id.visit( this, currentClassName );
    … // Do checks for non-static in static method
```

Question: do I need a private check here?

```
visitIdentifier( Identifier id, String ctx ) ::=
    id.decl = findDeclaration( ctx, id );
```

The mysteries of RefExpr

Recall Fail351.java



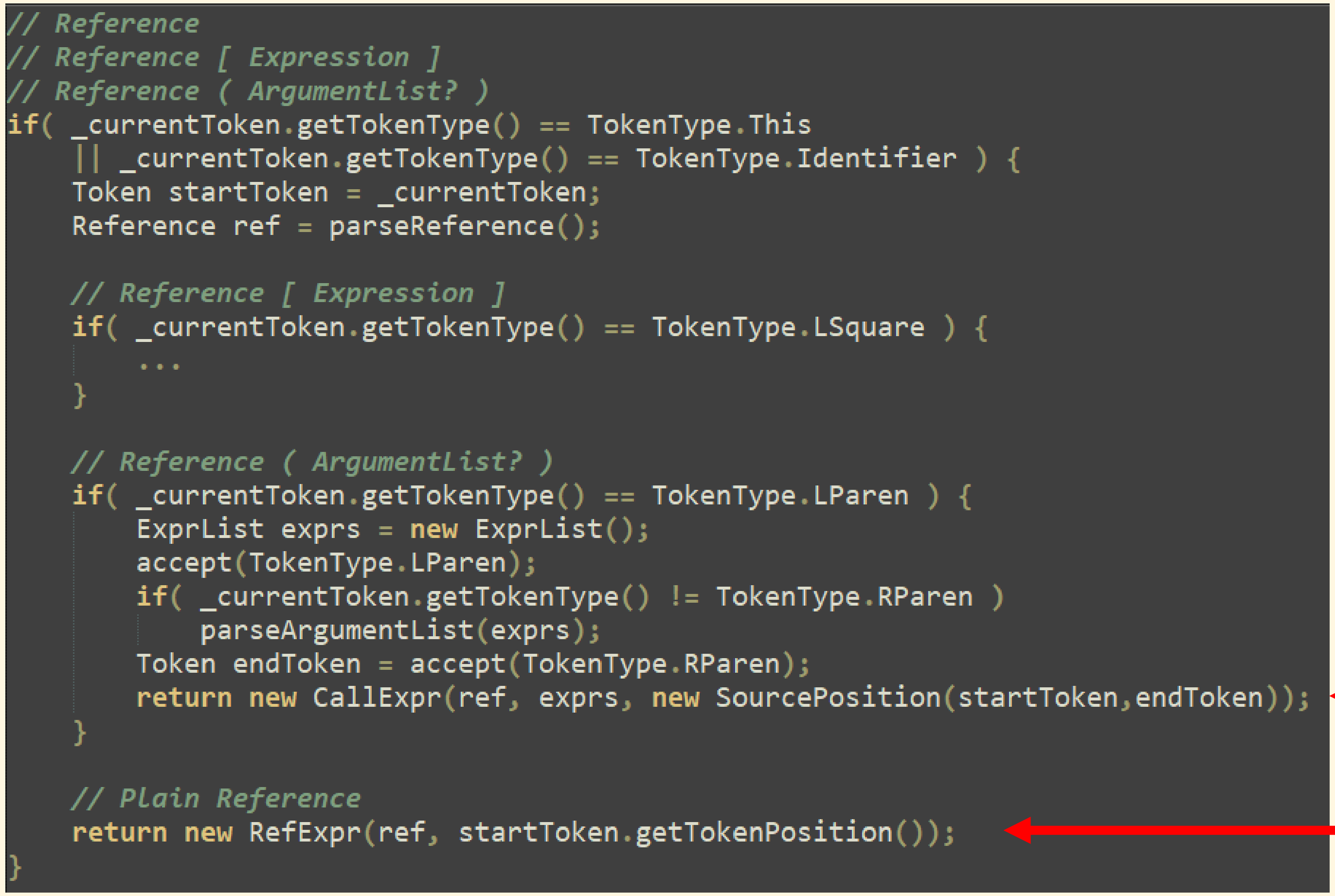
```
class TestClass {
    public static void staticContext() {
        int x = 0;
        x = pubfn;
    }
    public static int pubfn() { return 1; }
}
```

Method, not variable.

Recall: CallExpr / CallStmt

- Grammar:
  - CallStmt ::= Reference (ArgumentList?) ;
  - CallExpr ::= Reference (ArgumentList?) ;
  - RefExpr ::= Reference ;

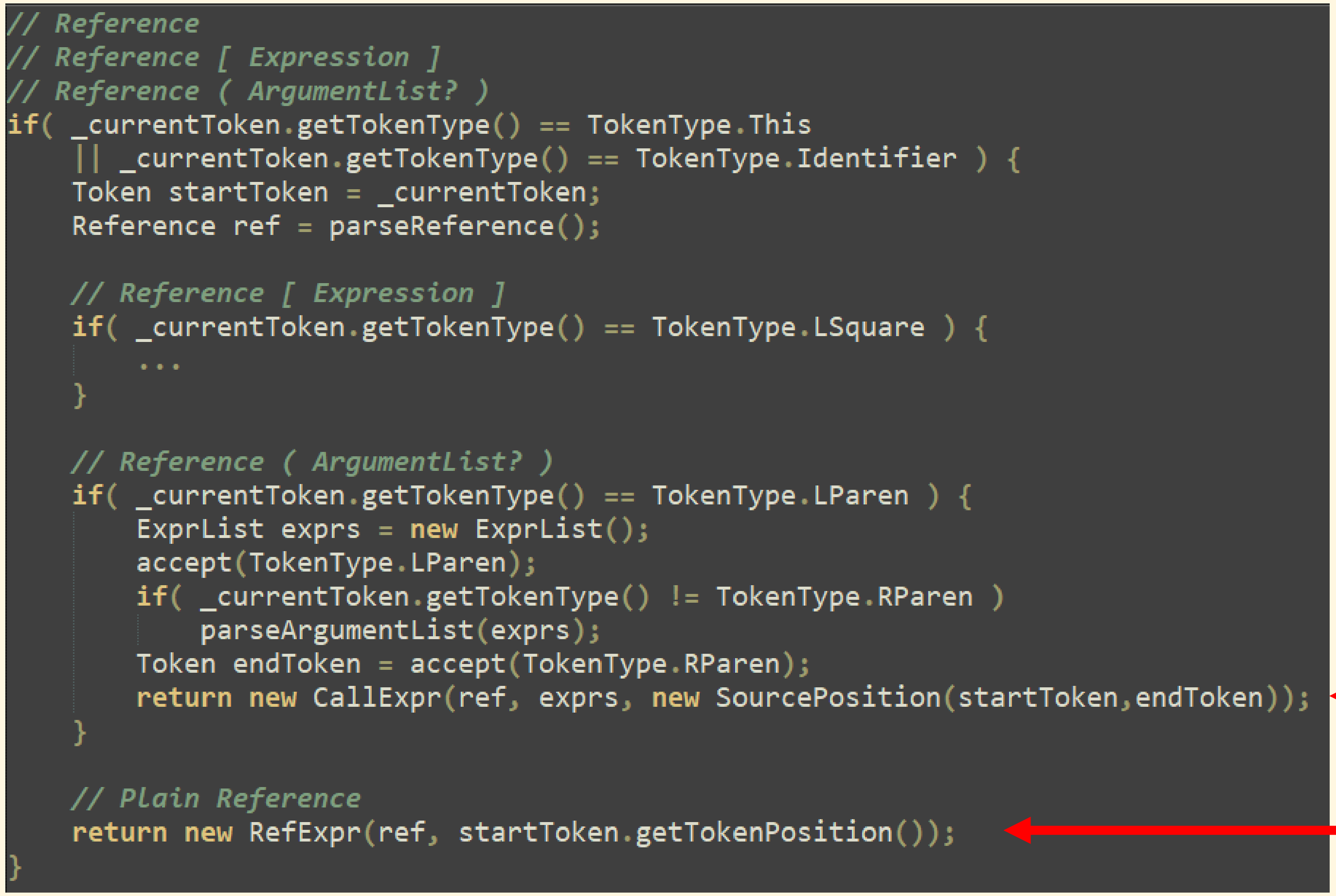
You did this in PA2



```
// Reference
// Reference [ Expression ]
// Reference ( ArgumentList? )
if( _currentToken.getTokenType() == TokenType.This
    || _currentToken.getTokenType() == TokenType.Identifier ) {
    Token startToken = _currentToken;
    Reference ref = parseReference();

    // Reference [ Expression ]
    if( _currentToken.getTokenType() == TokenType.LSquare ) {
        ...
    }

    // Reference ( ArgumentList? )
    if( _currentToken.getTokenType() == TokenType.LParen ) {
        ExprList exprs = new ExprList();
        accept(TokenType.LParen);
        if( _currentToken.getTokenType() != TokenType.RParen )
            parseArgumentList(exprs);
        Token endToken = accept(TokenType.RParen);
        return new CallExpr(ref, exprs, new SourcePosition(startToken,endToken));
    }

    // Plain Reference
    return new RefExpr(ref, startToken.getTokenPosition());
}
```

Instantiate a CallExpr vs RefExpr

You did this in PA2 (2)



```
// Reference
// Reference [ Expression ]
// Reference ( ArgumentList? )
if( _currentToken.getTokenType() == TokenType.This
    || _currentToken.getTokenType() == TokenType.Identifier ) {
    Token startToken = _currentToken;
    Reference ref = parseReference();

    // Reference [ Expression ]
    if( _currentToken.getTokenType() == TokenType.LSquare ) {
        ...
    }

    // Reference ( ArgumentList? )
    if( _currentToken.getTokenType() == TokenType.LParen ) {
        ExprList exprs = new ExprList();
        accept(TokenType.LParen);
        if( _currentToken.getTokenType() != TokenType.RParen )
            parseArgumentList(exprs);
        Token endToken = accept(TokenType.RParen);
        return new CallExpr(ref, exprs, new SourcePosition(startToken,endToken));
    }

    // Plain Reference
    return new RefExpr(ref, startToken.getTokenPosition());
}
```

DISJOINT CASES

Thus, PA3 check becomes easier.

- If it **is** a RefExpr, then it is **not** a CallExpr.
- Therefore, `visitRefExpr`, if the reference's declaration is a `MethodDecl`, then it is wrong.
  - E.g. `x = someFn;` instead of `x = someFn();`

- In a `CallExpr`, if the reference's declaration is NOT a `MethodDecl`, then it is wrong.
  - E.g. `int x = 0; int y = 0; y = x();`

The mysteries of QualRef

Figuring out QualRef

Left-hand-side	Returned Declaration after a Visit	Notes	Notes 2
Identifier: "CLASSNAME"	Declaration: `ClassDecl`	RHS must be a static ??	If the RHS is declared in a different class, RHS cannot be private.
Identifier: "this"	Declaration: `ClassDecl`	RHS must be a ??	
Identifier: "NAME" (not a class)	Declaration: `MemberDecl` or `LocalDecl`	RHS must be a ??	

First question: An identifier on the RHS of a QualRef is at what level of SI?

Figuring out QualRef

Left-hand-side	Returned Declaration after a Visit	Notes	Notes 2
Identifier: "CLASSNAME"	Declaration: `ClassDecl`	RHS must be a **static** `MemberDecl`	If the RHS is declared in a different class, RHS cannot be private.
Identifier: "this"	Declaration: `ClassDecl`	RHS must be a `MemberDecl`	
Identifier: "NAME" (not a class)	Declaration: `MemberDecl` or `LocalDecl`	RHS must be a `MemberDecl`	

Figuring out QualRef

Left-hand-side	**Returned Declaration after a Visit**	**Notes**	**Notes 2**
Identifier: "CLASSNAME" IdRef / Id	Declaration: `ClassDecl`	RHS must be a **static** `MemberDecl`	If the RHS is declared in a different class, RHS cannot be private.
Identifier: "this" ThisRef	Declaration: `ClassDecl`	RHS must be a `MemberDecl`	
Identifier: "NAME" (not a class)	Declaration: `MemberDecl` or `LocalDecl`	RHS must be a `MemberDecl`	

Translate this table into code. (Cleanup required)

1) If LHS.decl instanceof ClassDecl, then RHS must be static
   unless LHS instanceof ThisRef
2) Resolve RHS in the context of LHS
   2a) If LHS.decl is ClassDecl, easy, ctx is that class's name
   2b) If LHS.decl is MemberDecl, get that member's classname
   2c) If LHS.decl is LocalDecl, easy, ctx is that class type's name.
3) RHS.decl is always instanceof MemberDecl

Additional checks:
4) LHS cannot be a MethodDecl (miniJava shortcut)
5) If RHS is private, check if current class name equals their class
6) LHS must be a ClassType (classes have fields, everything else does not) if it isn't a ThisRef.
   6a) Thus, visit the ClassType to get the ClassDecl of the LHS.

How does this help in PA3?

PA3 Summarized

- Make Stack<HashMap<String,Declaration>> SI;
- Create level 0 and 1 immediately
- Forcibly inject predefined names into level 0 and 1
  - **Level 0:** String, _Printstream, System
  - **Level 1:** _Printstream.println, System.out\

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

PA3 Summarized (2)

- Forcibly inject all Classes and Fields into level 0 and 1
  - Is this class name used at level 0? Error, or add the class name to level 0. Why?
  - Is this field name used at level 1? Error, or add the name to level 1 (with context). Why?
- Begin visiting classes

PA3 Summarized (3)

- Begin visiting classes.
- Visit everything (e.g. `visitArrayType` visits the element type, and `MethodDecl` visits internal `Statement`s, etc.)
  - `visitClassDecl`: Name already added, just visit methods.
  - `visitMethodDecl`: Name already added, just visit statements.
  - `visitParameterDecl`: Does it already exist at **level 2+**?

Why?

PA3 Summarized (4)

- `visitClassType`: The identifier better be a class name!
    - Why?
- `visitBaseType`: do nothing.
    - Why?
- `visitArrayType`: visit the element type.
    - Why?

PA3 Summarized (5)

- `visitVarDecl`: Does the name exist at **2+**?
  - Question: How can I only check 2+?
  - Error or add at top scope level

- `visitCallStmt`: Just to reiterate as an example, visit all of the expressions in the Call *(visit everything!).*
  Visit the Reference too, and that reference better be have a methodDecl.

PA3 Summarized (6)

- `visitThisRef`: Return "CurrentClassDecl"
  - How can we keep track of the currentClassDecl?

- `visitIdRef`: Are we in a static method? If so and if the decl is a MemberDecl (level 1), make sure the IdRef's identifier's declaration is static and not currently being declared. (IdRef means access local class fields or locals)

PA3 Summarized (7)

- `visitRefExpr`: A reference expression (keyword: **expression**) cannot just have a reference be a class name.
  - Is the reference.decl a **ClassDecl** (and ref not instanceof ThisRef), then error. (E.g. x = CLASSNAME;)
  - Similarly, **RefExpr** cannot be a MethodDecl, otherwise it *should* have been a **CallExpr** not **RefExpr**.
    - E.g., `x = method;` *vs* `x = method();`
- `visitCallExpr`: Like CallStmt.
- `visitQRef`: Do Slide 44 check.
- Etc.

PA3 Type-Checking

- Every useful node returns a TypeDenoter.
- Two types of type-checks:
  - Are these two types the same?
  - Given two types and an operator, what is the result type?

- Example: At every visit method, ask,
  "What needs to be ensured here?
  What MUST be a ClassType?
  What MUST be an ArrayType (IxExpr)?
  What MUST be the same type (AssignStmt)?"

Midterm 2 Review

Midterm 2 is shorter

- Because of delays, it doesn’t make sense to have a full midterm this late in the semester.
- Instead, we made the midterm shorter by a question (and arguably easier).

- Your main goals should be to learn PA4, which is worth more than the midterm.

Optimization

- Dataflow / Codeflow analysis is not on the midterm.

- Key takeaway from optimization:
  - You don’t need to respect the input code, can generate more optimized code.

Byte Encoding / ModRMSIB

- You already learned how to do this in earlier prerequisite classes.
  - PA4 is new because of CISC, but it isn’t Midterm-worthy.

- Don’t worry about encoding bytes.

What should you know?

- How to generate “equivalent” code
- How to do identification and type-checking
- Midterm is cumulative with PA1 and PA2

- You should have some general mastery over what you have done on your compiler project

Side notes about PA4

- If “int is 4 bytes” is giving you trouble, just make everything 8 bytes and fix it later.
- Some PA4 starter files have “TODO” marked but won’t flag a compiler error. Easiest way to handle this: Find the word “TODO” in the starter files.

Feedback I’ve gotten thus far

- “I’m trying to compile a simple println program and that is dope”
  - Response: just imagine what it will be like when it is fully functional! (Imagine the stuff you can do with the compiler too)
- “A lot of the code isn’t mine, how can I attribute this?”
  - Response: the starter code is boring, you are implementing the more interesting stuff. Imagine adding a ton of line breaks and claiming you do a lot of LOCs. YOU control the interesting parts.

Questions?

- Let’s go over how the midterm is used
- Let’s go over the cheat sheet

End