Object-Oriented Programming

CSCI-UA 0470-001
Class 20
Instructor: Randy Shepherd

Scope

What is Scope?

- Scope is the area of visibility of identifiers such as variables, fields, methods, etc.

- For example, all the fields and methods of a class are visible within the class.

- Java & C++ have ‘static’ scoping rules.

- Also called ‘lexical scope’, based on structure of program text

- (There is another type called ‘dynamic scoping’, which is uncommon in modern languages.)

Static Scoping

- To connect a name to a variable, the compiler must find the declaration
- Search process
    1. Search declarations locally
    2. Then next largest surrounding scope
    3. Repeat 2 until declaration for the name is found
- Enclosing static scopes are called its ‘static ancestors’
- The nearest static ancestor is called a ‘static parent’

Why Scope?

- Scopes allow us to have the same name in different blocks of a program because of the concept of scope.

- Scopes are like enclosing walls around sets of names. We can declare the same name within different scopes because as soon as execution leaves the confines of one scope, it no longer "knows" the names from that scope.

```java
public class Scoping {  // class scope begins

    // Because 'a' is declared just inside the *class curly braces* it is in 'class scope'
    // therefore its visible everywhere inside the class. This is the 'widest' scope in this file.
    private int a = 0;

    public void methodOne() {  // methodOne scope begins
        // Because 'b' is declared inside the *methodOne curly braces*
        // its only visible in methodOne
        int b = 0;
        if (a == b) { // conditional scope begins
            // Because 'c' is declared inside conditional curly braces its only
            // visible inside the conditional
            int c = 1;
            // Because 'b' is declared in a scope which surrounds the scope of
            // the conditional, 'b' is visible.
            b += c;
            // Because 'a' is declared in a scope that surrounds this scope of
            // the conditional, 'a' is visible.
            a += c;
        }
        // 'c' is no longer visible
    }
    // 'b' no longer visible

    public void methodTwo() { // methodTwo scope begins
        int d = a;
        // 'a' is is visible since its in a scope that surrounds this scope (class scope)
        // 'b' & 'c' not visible, they are in a scope that does not surround the scope of this method
        // 'd' only visible inside methodTwo
    }
    // 'd' no longer visible

    // Note: Constructors obey the same rules as methods w.r.t. variable scoping
}
```

Symbol Table

Symbol Table

- In a compiler, after ASTs have been constructed, a compiler must check whether the input program is type-correct.

- A compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program.

- Storing this information is this is the task of a symbol table.

Symbol Table Contents

- More specifically, a symbol table stores:
  - for each variable name, its type. If the variable is an array, it also stores dimension information.
  - for each function, its parameter list and its return type. Each formal parameter has name and type.
- All with some awareness of the *scope rules* of the language in question.

In the Translator

- We will use a symbol table to resolve what implementation of an overloaded method is referenced at some call site.

- In other words, based on the types of the *arguments* in a method invocation, we can “match” that invocation with the correct version of an overloaded method, based on the *parameter* types.

- We use the symbol table when “matching”.

Overload Resolution

- When resolving some call site to a method declaration, if more than one is both accessible and applicable, choose the most specific method.

```
1  private String m(int i) {
2      System.out.println("m(int)      : " + i);
3      return "int";
4  }
5
6  public String m(long l) {
7      System.out.println("m(long)     : " + l);
8      return "long";
9  }
10
11 public String m(Object o) {
12     System.out.println("m(Object)   : " + o);
13     return "Object";
14 }
15
16 public static String m(A a) {
17     System.out.println("m(A)        : " + a);
18     return "A";
19 }
20
21 private String m(B b) {
22     System.out.println("m(B)        : " + b);
23     return "B";
24 }
```

Overload Resolution

- What things do we need to know to resolve this call?

```
// Assuming B extends A
o.m(new B());
```

```
1  private String m(int i) {
2      System.out.println("m(int)     : " + i);
3      return "int";
4  }
5
6  public String m(long l) {
7      System.out.println("m(long)    : " + l);
8      return "long";
9  }
10
11 public String m(Object o) {
12     System.out.println("m(Object)  : " + o);
13     return "Object";
14 }
15
16 public static String m(A a) {
17     System.out.println("m(A)       : " + a);
18     return "A";
19 }
20
21 private String m(B b) {
22     System.out.println("m(B)       : " + b);
23     return "B";
24 }
```

Overload Resolution

- What things do we need to know to resolve this call?

```
// Assuming B extends A
o.m(new B());
```

- The type of the ‘o’
- The type of ‘new B()’
- That m(B) is private
- That B extends A
- That there is a m(A)

```
1  private String m(int i) {
2      System.out.println("m(int)     : " + i);
3      return "int";
4  }
5
6  public String m(long l) {
7      System.out.println("m(long)    : " + l);
8      return "long";
9  }
10
11 public String m(Object o) {
12     System.out.println("m(Object)  : " + o);
13     return "Object";
14 }
15
16 public static String m(A a) {
17     System.out.println("m(A)       : " + a);
18     return "A";
19 }
20
21 private String m(B b) {
22     System.out.println("m(B)       : " + b);
23     return "B";
24 }
```

Xtc's Symbol Table

- We need a symbol table to answer these questions.
- Xtc has one. We just need to learn how to use it.
- We'll start today.

Caveat

- The symbol table builder you are seeing today is not complete. However, it is sufficient for you to get started and to learn from.
- Xtc has a symbol table builder for Java. It is a nightmare to use.
- I am writing some heroic code to make using it easier. More on this soon…

Building the Symbol Table

- In order to have a symbol table for some source program, we need to instruct Xtc on how to build it.

- We do this with a visitor specific to this purpose.

```
public void visitClassDeclaration(GNode n) {
    String className = n.getString(1);
    table.enter(className);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitMethodDeclaration(GNode n) {
    String methodName = JavaEntities.methodSymbolFromAst(n);
    table.enter(methodName);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitBlock(GNode n) {
    table.enter(table.freshName("block"));
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitForStatement(GNode n) {
    table.enter(table.freshName("forStatement"));
    table.mark(n);
    visit(n);
    table.exit();
}
```

Building the Symbol Table

- For each GNode that creates a scope, we need to add that scope explicitly to a symbol table.

- ‘table’ is an instance of *xtc.util.SymbolTable*

```
public void visitClassDeclaration(GNode n) {
    String className = n.getString(1);
    table.enter(className);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitMethodDeclaration(GNode n) {
    String methodName = JavaEntities.methodSymbolFromAst(n);
    table.enter(methodName);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitBlock(GNode n) {
    table.enter(table.freshName("block"));
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitForStatement(GNode n) {
    table.enter(table.freshName("forStatement"));
    table.mark(n);
    visit(n);
    table.exit();
}
```

Building the Symbol Table

- Therefore for each scope-creating GNode, we visit that GNode and "enter" the scope.

- This creates a scope in the SymbolTable

```
public void visitClassDeclaration(GNode n) {
    String className = n.getString(1);
    table.enter(className);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitMethodDeclaration(GNode n) {
    String methodName = JavaEntities.methodSymbolFromAst(n);
    table.enter(methodName);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitBlock(GNode n) {
    table.enter(table.freshName("block"));
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitForStatement(GNode n) {
    table.enter(table.freshName("forStatement"));
    table.mark(n);
    visit(n);
    table.exit();
}
```

Building the Symbol Table

- Once inside a scope in the table, we “mark” the node.
- This associates the GNode with a scope in the table, making lookups easier.
- Next we call visit, descending the children of this scope-introducing node.

```
public void visitClassDeclaration(GNode n) {
    String className = n.getString(1);
    table.enter(className);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitMethodDeclaration(GNode n) {
    String methodName = JavaEntities.methodSymbolFromAst(n);
    table.enter(methodName);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitBlock(GNode n) {
    table.enter(table.freshName("block"));
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitForStatement(GNode n) {
    table.enter(table.freshName("forStatement"));
    table.mark(n);
    visit(n);
    table.exit();
}
```

Building the Symbol Table

- Then for each child of the scope-introducing node..
- fill that scope in the table with identifiers and type information.
- The code required for this is pretty ugly.

```
public final Type visitFormalParameter(final GNode n) {
    assert null == n.get(4) : "must run JavaAstSimplifier first";
    String id = n.getString(3);
    Type dispatched = (Type) dispatch(n.getNode(1));
    Type result = VariableT.newParam(dispatched, id);
    if (n.getGeneric(0).size() != 0)
        result.addAttribute(JavaEntities.nameToModifier("final"));
    if (null == table.current().lookupLocally(id)) {
        table.current().define(id, result);
        result.scope(table.current().getQualifiedName());
    } else
        runtime.error("duplicate parameter declaration " + id, n);
    assert JavaEntities.isParameterT(result);
    return result;
}
```

Building the Symbol Table

- When done recursing into the children and extracting type information, we exit the scope.

- *Key intuition:* The table is always ‘in a scope’ at any given moment based on the series of entrances and exits performed.

```
public void visitClassDeclaration(GNode n) {
    String className = n.getString(1);
    table.enter(className);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitMethodDeclaration(GNode n) {
    String methodName = JavaEntities.methodSymbolFromAst(n);
    table.enter(methodName);
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitBlock(GNode n) {
    table.enter(table.freshName("block"));
    table.mark(n);
    visit(n);
    table.exit();
}

public void visitForStatement(GNode n) {
    table.enter(table.freshName("forStatement"));
    table.mark(n);
    visit(n);
    table.exit();
}
```

Using the Symbol Table

- Later, in another phase, we can traverse the tree and do lookups of identifier information.

- Again, the table itself has a notion of scope.

- While inside a scope in the table you have access to the identifiers in that scope.

```
public void visitForStatement(GNode n) {
    SymbolTableUtil.enterScope(table, n);

    runtime.console().p("Entered scope ")
      .pln(table.current().getName());

    visit(n);

    SymbolTableUtil.exitScope(table, n);
}

public void visitPrimaryIdentifier(final GNode n) {
    String name = n.getString(0);

    if (table.current().isDefined(name)) {
        Type type = (Type) table.current().lookup(name);
        if (JavaEntities.isLocalT(type))
            runtime.console().p("Found local variable ")
              .p(name).p("of type").pln(type.tag().name());
    }
}
```

Team Exercise

- The best way to understand it is to start using it.
- Get into your teams. You know the drill.
- The repo https://github.com/nyu-oop/symbol-table-in-class