Program Representations & Transformations

Outline

- Java Bytecode Review
- Abstract Syntax Trees
- Program Transformations
  - Bytecode
  - AST

Review: Java Bytecode

Each line of source corresponds to many low level operations

What's in a class?

When we compile, we get a class file, this is filled with hex "opcodes"

Using javap, we can translate those opcodes into something more human readable

Review: Java Bytecode

```
public class Ex0 {

   public int add(int x) {
      return x + 1;
   }

   public void start() {
      int z = add(1);
   }
}
```

```
public int add(int);
   Code:
      0: iload_1
      1: iconst_1
      2: iadd
      3: ireturn

   LocalVariableTable:
     Start  Length  Slot  Name
        0       4     0  this   LEx0;
        0       4     1     x   I

  public void start();
   Code:
      0: aload_0
      1: iconst_1
      2: invokevirtual #2   // Method add:(I)I
      5: istore_1
      6: return

   LocalVariableTable:
     Start  Length  Slot  Name
        0       7     0  this   LEx0;
        6       1     1     z   I
```

Operand Stack

Local Variable Table

Example 2

```
public class Ex1 {

    public void foo(int x) {
        int y = x + 1;
    }

    public void baz() {
        int z = 99;
        foo(z);
    }
}
```

```
public void foo(int);
  Code:
    0: iload_1
    1: iconst_1
    2: iadd
    3: istore_2
    4: return

  LocalVariableTable:
    Start  Length  Slot  Name
       0       5     0  this   LEx1;
       0       5     1     x   I
       4       1     2     y   I

public void baz();
  Code:
    0: bipush        99
    2: istore_1
    3: aload_0
    4: iload_1
    5: invokevirtual #2   // Method foo:(I)V
    8: return

  LocalVariableTable:
    Start  Length  Slot  Name
       0       9     0  this   LEx1;
       3       6     1     z   I
```

Operand Stack

Local Variable Table

Method Descriptors - Match the method with its descriptor

```
1.  int foo(int x, int y)
2.  void bar(boolean myFlag)
3.  boolean isOpen(File f)
4.  String getName(Student s)
5.  int baz(Object o, int[] z)
```

```
a.  (Ljava/lang/Object;[I)I;
b.  (Ljava/lang/String;)LStudent;
c.  ()V
d.  (Z)V
e.  (Ljava/io/File;)Z
f.  (II)I
g.  (I)Ljava/io/File;
h.  ()[I
i.  (LStudent;)Ljava/lang/String;
j.  ([I[I)V
k.  (Ljava/lang/Object;)V
```

Solution: 1f, 2d, 3e, 4i, 5a

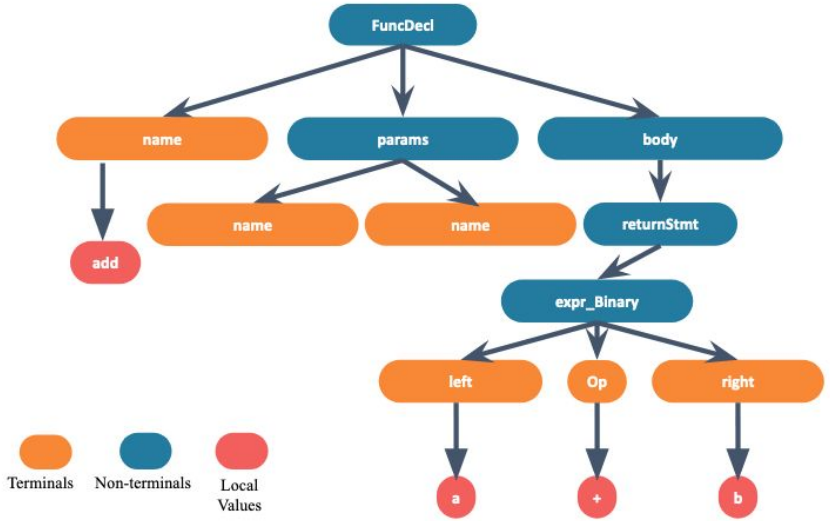
Program Representation 2: Abstract Syntax Trees

Representing Code as a Tree

Code has a natural hierarchical structure which cannot be captured as plain text

Classes have methods, methods have statements...

```
function add(a, b) {  return a + b; }
```



AST Explorer

https://astexplorer.net/

Java - `Statements` vs `Expressions`

- A statement is a complete unit of execution that performs an action. It often ends with a semicolon
- An expression is a combination of variables operators, iterals, and method calls that evaluates to a single value

Java - `Statements` vs `Expressions`

- int x = 5;
  - Statement
- 2 + 3
  - Expr
- System.out.println(“Hello World”);
  - Statement
- Math.sqrt(9)
  - Expr
  - OR expression-statement

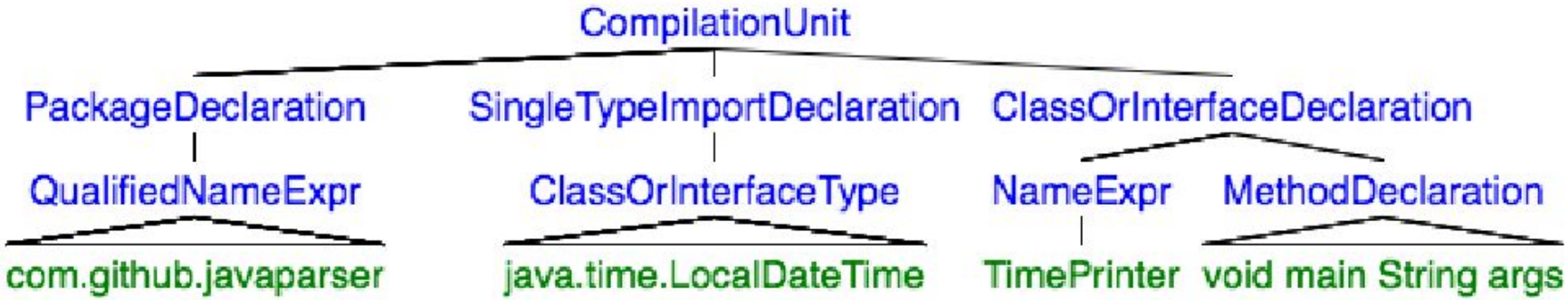
Java AST - node types

```java
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {

    public static void main(String args[]){
        System.out.print(LocalDateTime.now());
    }
}
```



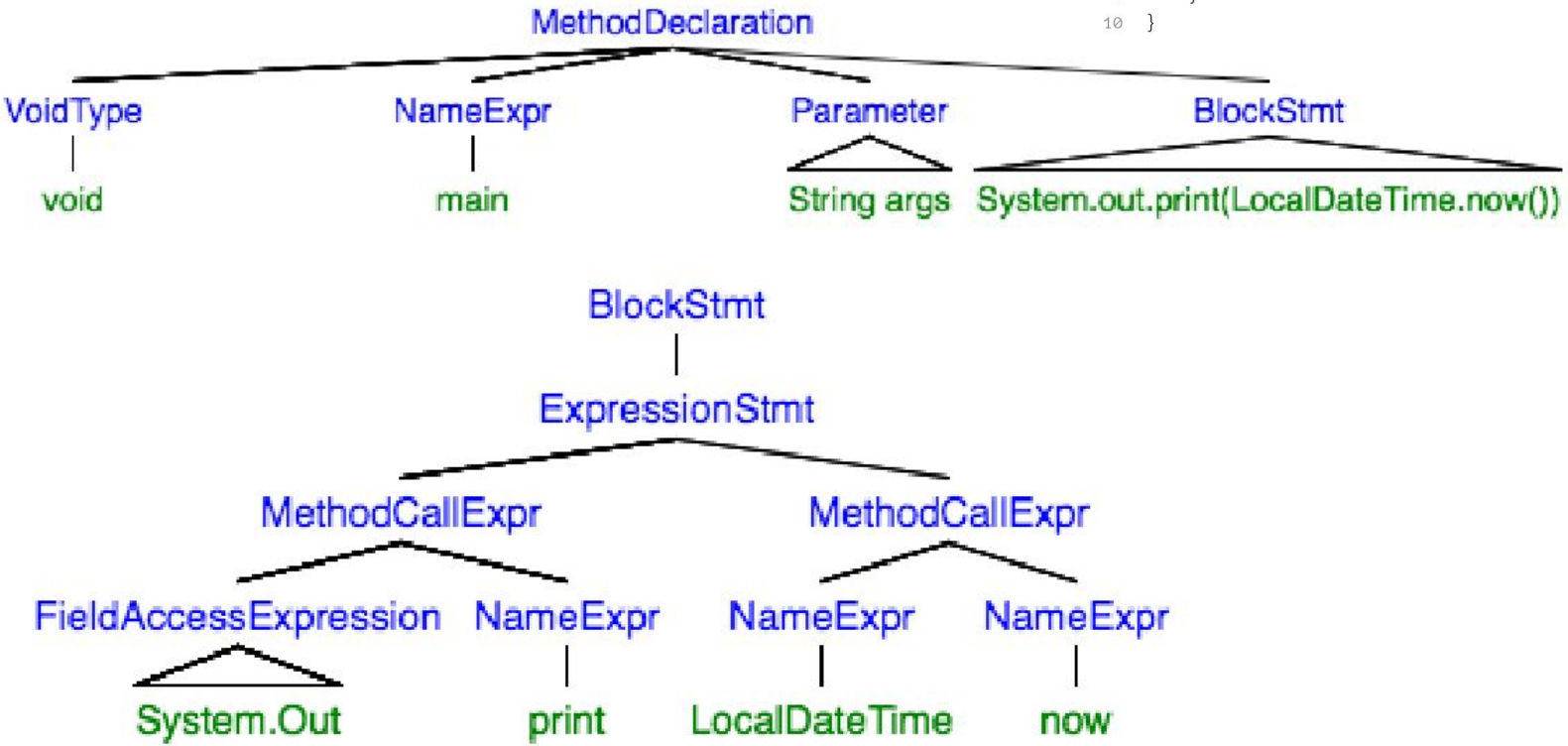
Java AST - node types

```java
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {

    public static void main(String args[]){
        System.out.print(LocalDateTime.now());
    }
}
```

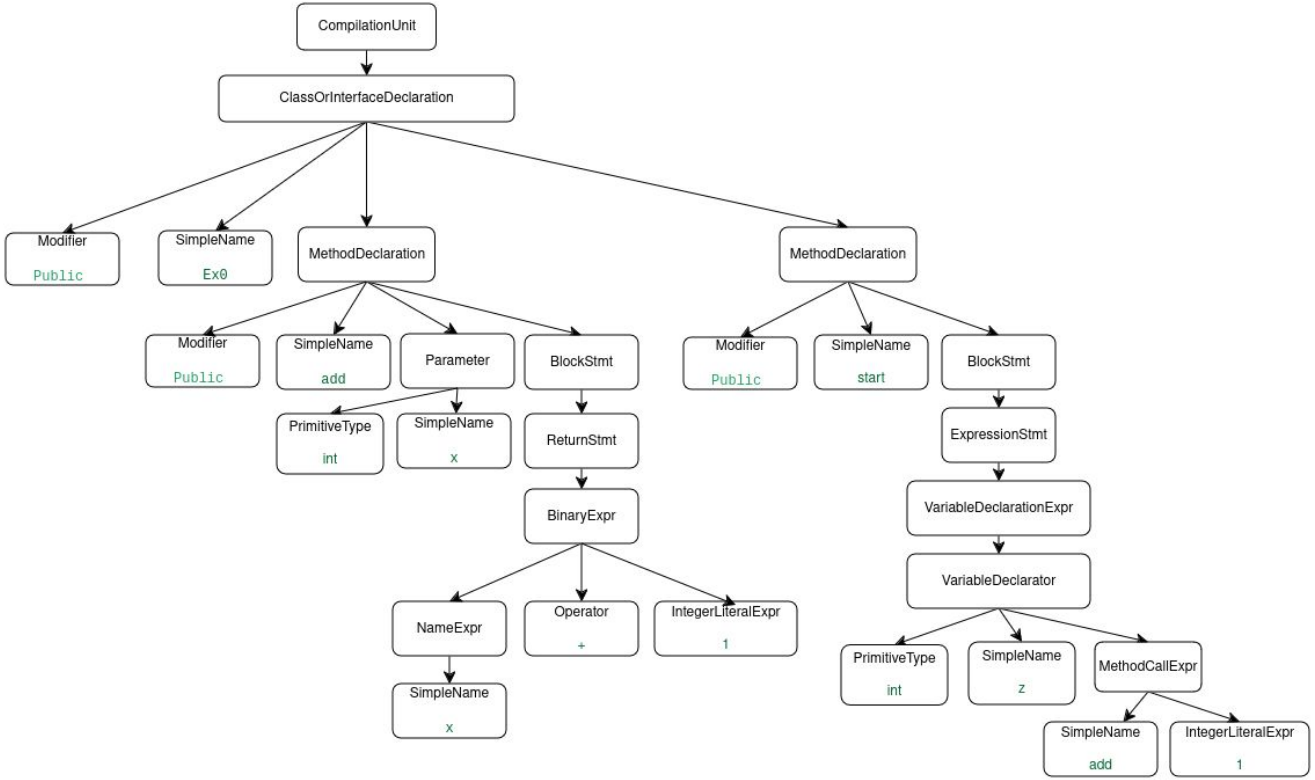


What would the AST look like?

```
public class Ex0 {

    public int add(int x) {
        return x + 1;
    }

    public void start() {
        int z = add(1);
    }
}
```



Program Transformations

Why might we want to transform a program?

- Optimizations
- Obfuscations
- Debugging
- Performance Monitoring
- ...
- We’ll learn more in this course!

AST Transformations

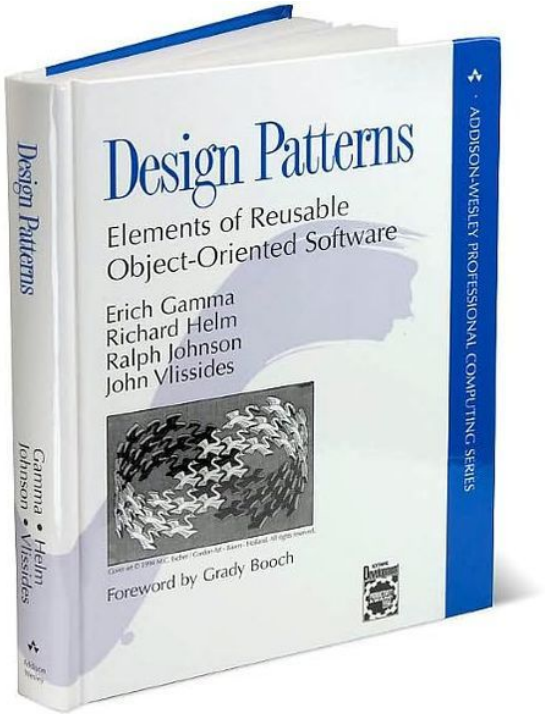
AST Transformations

Example: inject performance logging for each method (print time taken for each method)

- Find all method declarations
- Create new statements
  - Store start time
  - Store end time
  - Print end - start
- Insert them into the method declaration
  - In the proper location!

Traversing the AST tends to be laborious and error prone. This is largely due to the process relying on recursion and frequent type checking to attain your goal

Design Patterns



- Reusable solutions to common software design problems
- Provide a way to structure your code in a way that is flexible, maintainable, and efficient
- *“All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects.”*

Visitor Design Pattern

- **Visitor Design Pattern** commonly used for tree transformations
- Allows you to separate the operations / transformations from the traversal process
- Trees often have different types of nodes. The Visitor pattern makes it easy to define different operations for each node type by using method overloading in the visitor.
- Each operation is encapsulated in a "visitor," which visits different node types (like addition or multiplication) and processes them.

JavaParser Library

- Allows you to interact with Java source code as an AST

- Provides mechanism to navigate the tree with *Visitor Support*
  - Allows you to write operations without needing to write traversal code

A Simple Visitor

Let’s create a visitor to print the name of each method in a program

You can also pass a value to the visitor!

- Useful to save or load state

A Simple Modifying Visitor

Let's write a visitor to change variable names (declarations and references) to all caps in Method decls

```
public class Vars {

   public void decls() {
      int foo = 15;
      int helloworld = 20;
      int added = foo + helloworld;
   }


   public void dontchange() {
      int lower = 15;
      int stillLower = 10;
      int lowercase = lower + stillLower;
   }
}
```

JavaParser library

Basic Workflow:

1. Parse the source into an AST
   a. JavaParser.parse(sourceCode);
2. Transform the AST
   a. Using a Visitor, traverse the AST and perform a transformation
3. Write the modified AST
   a. Call toString() on the AST troot (compilationUnit) o get the source

Bytecode Transformations

ASM

Java Library for Bytecode manipulation

Basic Workflow:

1. Read the bytecode
    a. ClassReader

2. Transform the bytecode

3. Write the modified bytecode
    a. ClassWriter

Bytecode Transformation

Increment every constant by 1 in the main method

```
public class Constants {

    public static void dontModify(int q) {
        int x = 100;
        int y = 10;
        int z = x + y + 1;
    }

    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int c = 30;

        dontModify(100);
    }

}
```

```
public static void dontModify(int);
  Code:
     0: bipush        100
     2: istore_1
     3: bipush        10
     5: istore_2
     6: iload_1
     7: iload_2
     8: iadd
     9: iconst_1
    10: iadd
    11: istore_3
    12: return

public static void main(java.lang.String[]);
  Code:
     0: bipush        10
     2: istore_1
     3: bipush        20
     5: istore_2
     6: bipush        30
     8: istore_3
     9: bipush        100
    11: invokestatic  #7      // Method dontModify:(I)V
    14: return
```

Summary

- We learned how to transform programs over two different representations

- HW1 Released (due Feb 12)
  - AST and Bytecode transformations

- Lab1 due Sunday