2022

Abstract Syntax Trees (ASTs) and the Visitor Pattern

Raffi T. Khatchadourian
CUNY Hunter College

Programming Languages/Software Engineering

Abstract Syntax Trees (ASTs) and the Visitor Pattern

Abstract Syntax Trees (ASTs)

Intermediate Representations (IR)

- We learned last week that intermediate program representations (IR) facilitate analysis and transformation.
- 3-address code is great for **simplifying** a large and complicated language.
  - For example, one statement can be broken down to many “instructions.”
  - A language can have many kinds of statements but there’s a limited amount of instructions.
  - Conditionals in a high-level language like C++, Java, or Python:
    - if statement, if-else statement.
    - switch and case statements.
    - while statements, for loops, do-while.
  - Conditionals in 3-address IR.
    - Jump!

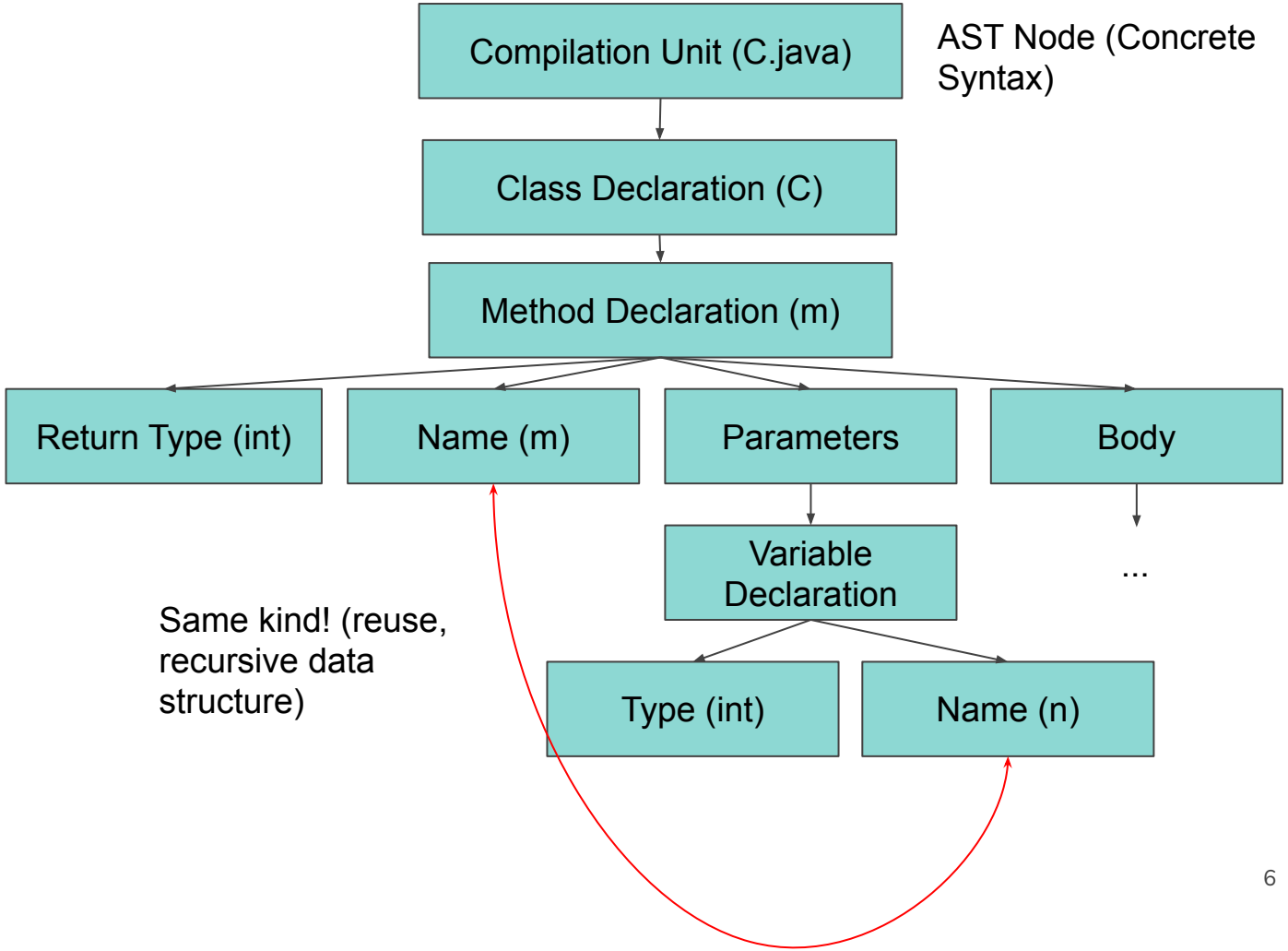
Abstract Syntax Trees (ASTs)

- 3-address code is convenient for *complicated* analyses but not so convenient for *source-to-source* transformation, type-checking, type-inference, bug finding, etc.
- Despite the numerous kinds of statements, sometimes, it's more convenient to stay at a high-level of representation. This allows you to:
  - Easily relate analysis results to the source.
  - Give developers feedback (error messages, etc.).
  - Transform the source.
- **NOTE**: If we just wanted to transform the bytecode (like an optimizing compiler), we can stay at the 3-address code IR level since it is very close to the byte (machine) code.

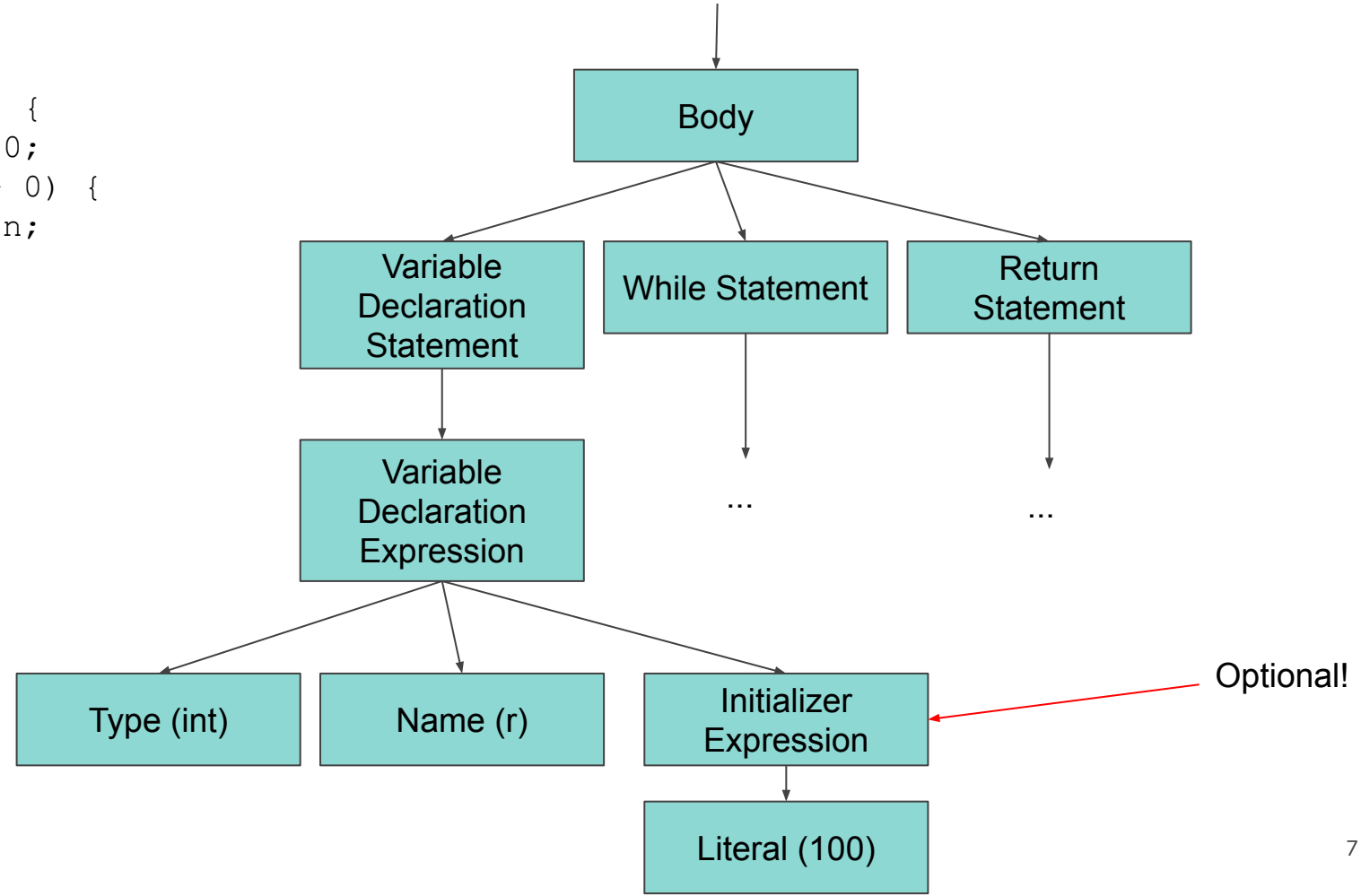
Abstract Syntax Trees (ASTs)

- Syntax trees are trees that represent program structure.
- Nodes depend on kind of tree, edges denote containment.
- Two kinds: *concrete* and *abstract*.
  - *Concrete* trees have a 1-1 mapping between a *particular* program.
    - Each node represents *specific* programming constructs (e.g., statements, expressions).
  - *Abstract* trees abstract away the details of particular constructs.
    - Each node represents the *kind* of programming construct rather than the actual syntax.
    - Can also associate information from the syntax (e.g., expression type). These are called *bindings* in some frameworks.
    - This is the one we'll use.

```
// C.java
class C {
  int m(int n) {
    int r = 100;
    while (n > 0) {
      r = r - n;
      --n;
    }
    return r;
  }
}
```



```
// C.java
class C {
  int m(int n) {
    int r = 100;
    while (n > 0) {
      r = r - n;
      --n;
    }
    return r;
  }
}
```



ASTs in the Eclipse SDK

- The Eclipse SDK (Software Development Kit) has a variety of APIs that process ASTs.
- Used extensively in language tools, (incremental) compiler, etc.
- AST API is specific per language.
- For Java, the JDT (Java Development Tools) has an AST type specific to Java.
  - There's a terrific Eclipse plug-in called AST View that allows you to view the AST for any Java program.
  - Other tools and documentation at https://khatchad.commons.gc.cuny.edu/research/background.
- See this article for examples.

```
// can build an AST from an existing project but, for simplicity, we'll just build one from a string.
Document document = new Document("import java.util.List;\nclass X {}\n");

ASTParser parser = ASTParser.newParser(AST.JLS3); // JLS3 stands for Java Language Specification 3.
parser.setSource(document.get().toCharArray()); // set the parser's source.

// Do the parsing. Can take a while, so there is an optional progress monitor parameter.
// A compilation unit is a source file (e.g., C.java). It is the root note of the tree.
CompilationUnit cu = (CompilationUnit) parser.createAST(null);

// this API also allows you to change the AST! This call will record any changes made to the node.
cu.recordModifications();

// every node has a method to retrieve the AST object it is related to. Useful for creating new nodes.
AST ast = cu.getAST();

ImportDeclaration id = ast.newImportDeclaration(); // make a new node.
id.setName(ast.newName(new String[] {"java", "util", "Set"});

cu.imports().add(id); // add import declaration at end

// don't actually change the source directly but instead return an object that contains "edits."
TextEdit edits = cu.rewrite(document, null);
UndoEdit undo = edits.apply(document); // can be undone.
```

Visitor Pattern

“Exploring” ASTs

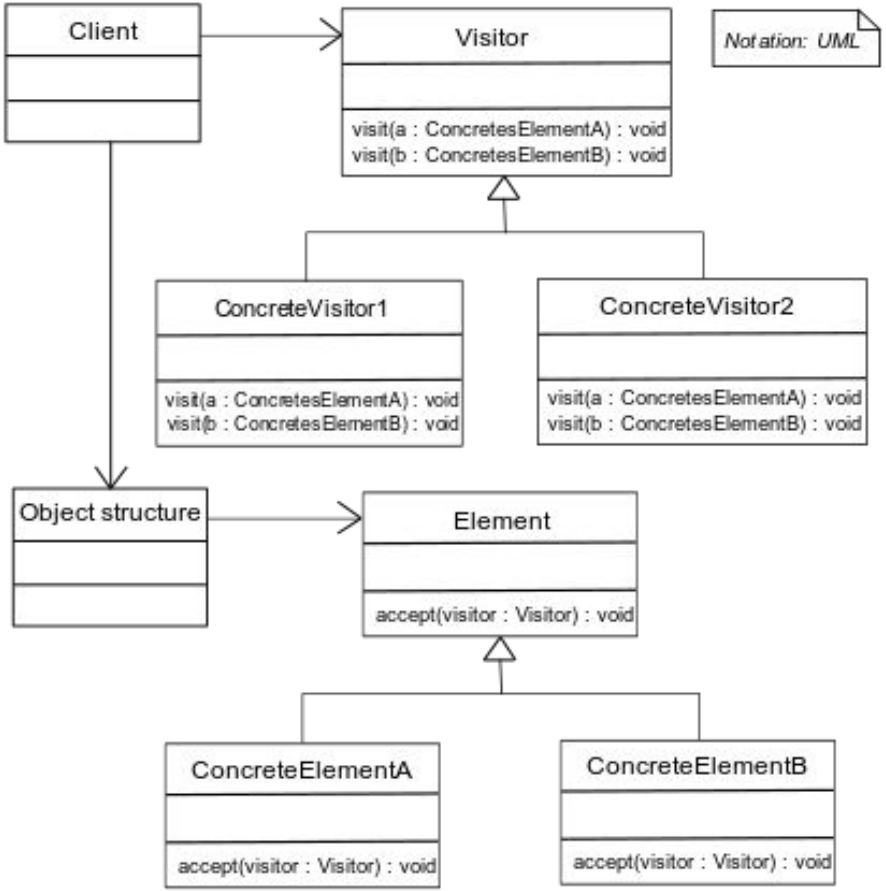
- Once you at a node, e.g., a compilation unit, can use API to either go up or down the tree. For example:
  - Go up via a call to getParent().
  - Go down depending on the type of node.
    - For example, a Java compilation unit might have a method like getTypeDeclarations().
    - A class declaration node may have a method getMethodDeclarations() or getFieldDeclarations().
    - An expression node may have a method getOperator().
- Can determine the type of node either using **instanceof** operator (e.g., if (node instanceof MethodDeclarationNode) or getNodeType() call (e.g., if (node.getNodeType() == ASTNode.METHOD_DECLARATION).
  - getNodeType() is convenient for (large) switch statements.

Visitor Pattern

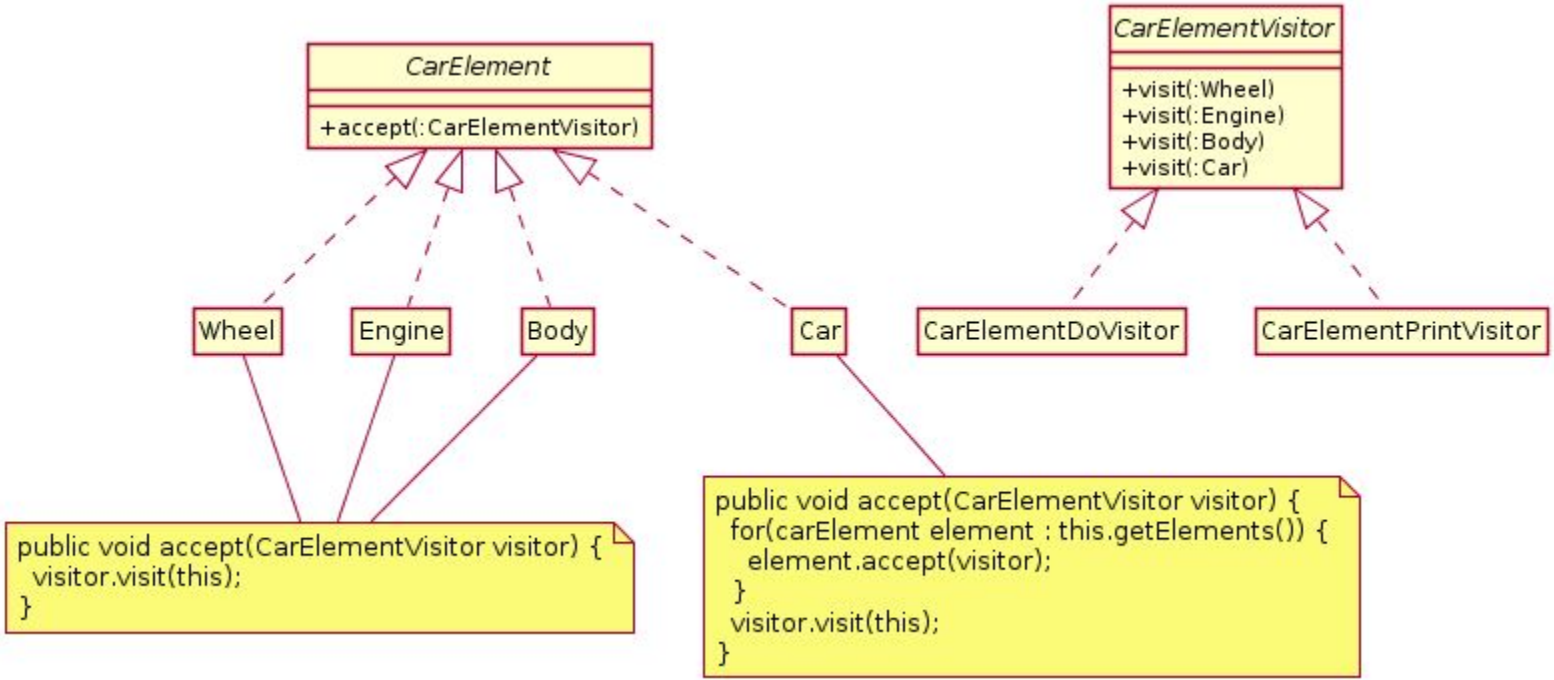
- Often times, you are interested in processing particular *kinds* of nodes.
- For example, you might want to visit all class declaration nodes to count the number of abstract classes whose name does not begin with the word “Abstract.”
- Java allows for multiple type declarations per file (compilation unit) as well as *nested* type declarations.
- Thus, it would be inconvenient to find all class declarations using the aforementioned API calls.
- Luckily, AST node types form a **type hierarchy**.
  - For example, there is a type called **Node** that is the parent of **all** AST node types.
  - As another example, a type like **VariableDeclarationStatement** is a subtype of **Statement**.
- This allows us to use **parametric polymorphism** to conveniently *visit* each node in an AST.
- This is called the *visitor pattern*.

Visitor Pattern

- Explained in Design patterns: elements of reusable object-oriented software under behavioral patterns.
- Makes extensive use of dynamic dispatch.
- Useful when:
  - You are interested in processing particular kinds of nodes.
  - Order in which nodes are visited is not important.
- Each AST node “accepts” a node visitor.
- A specific node visitor (one you create, for example) subclasses a **Visitor** type.
- Overrides methods with parameter types corresponding to the node types to be visited by the **specific** visitor.
- The specific visitor can also stop the entire process by returning **false** (at least in the JDK API).



Visitor Example UML



Visitor Example “Visitable” Interfaces

```
interface CarElement {
    void accept(CarElementVisitor visitor);
}

interface CarElementVisitor {
    void visit(Body body);
    void visit(Car car);
    void visit(Engine engine);
    void visit(Wheel wheel);
}
```

Visitor pattern. (2018, February 7). In *Wikipedia, The Free Encyclopedia*. Retrieved 16:11, February 7, 2018, from https://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=824451901

Visitor Example “Visitable” Types

```
class Car implements CarElement {
    CarElement[] elements;

    public Car() {
        this.elements = new CarElement[] {
            new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left"), new Wheel("back right"),
            new Body(), new Engine()
        };
    }

    public void accept(final CarElementVisitor visitor) {
        for (CarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}
```

Visitor Example “Visitable” Types

```
class Body implements CarElement {
    public void accept(final CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    public void accept(final CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor Example “Visitable” Types

```
class Wheel implements CarElement {
    private String name;

    public Wheel(final String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void accept(final CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor Example “Visitor” Types

```
class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(final Body body) {
        System.out.println("Visiting body");
    }

    public void visit(final Car car) {
        System.out.println("Visiting car");
    }

    public void visit(final Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(final Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }
}
```

Visitor Example “Visitor” Types

```
class CarElementDoVisitor implements CarElementVisitor {
    public void visit(final Body body) {
        System.out.println("Moving my body");
    }

    public void visit(final Car car) {
        System.out.println("Starting my car");
    }

    public void visit(final Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(final Engine engine) {
        System.out.println("Starting my engine");
    }
}
```

Visitor Example Driver

```
public class VisitorDemo {
    public static void main(final String[] args) {
        final Car car = new Car();

        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Visitor Example Output

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

Homework

- Read https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST.
- Assignment 1 assigned this week.
  - Make an Eclipse plug-in.
  - Analyze ASTs.
- Redo Assignment 0 part B using visitor pattern.
  - Provided by the WALA APIs.
    - **Don't** write your own pattern implementation!
    - **Do** write your own **visitor**.