Program Parser

Source Code Analysis

Munster Technological University

Dr Farshad Ghassemi Toosi

Code Analyzing/Inspecting

Physicians require tools and techniques to inspect and diagnose diseases (e.g., X-Ray, Tests etc.), so do Code analyzers.

1) Breaking the code-base down into components pieces.
2) Identifying the components (their type and roles).
3) Identifying the relations between components.
4) Identifying patterns in the source code.

The first three steps require a Program parser.

How automatically Parse a Programming Language

- Apply an existing library/tool that supports an specific language. E.g., Javaparser, JDT plug-ins (Java development tools), AST for Python and etc.
- Develop your own custom parser manually. E.g., regular expression (Regex) can be used to detect different syntax and etc. (Grammar that defines the language).
- Using tools or libraries to generate a parser. Similar to the last option, there are tools are developed/designed specifically for building a Parser for different languages. E.g., ANTLR, that you can use to build parsers for any language.
  - **APG**
  - **BYACC/J**
  - **CUP is the acronym of Construction of Useful Parsers**

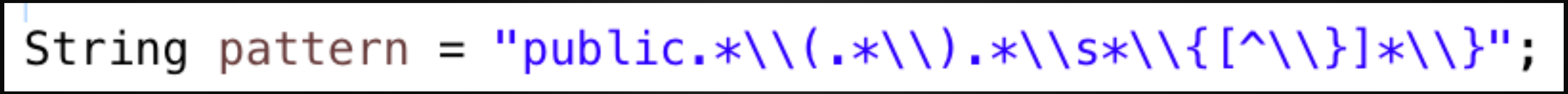
This option is used when particular need is required.

A proper reference can be found here.
https://tomassetti.me/parsing-in-java/

Manual Custom Parser

String manipulation (e.g., regex).

Exercise 1: Create a small program that can detect only public methods with any returned type and name in a given Java class.



```
String pattern = "public.*\\(.*\\).*\\s*\\{[^\\}]*\\}";
```

JavaParser

- Open source and free Java parser.
- The most popular parser for the Java language.
- Active supporting community.

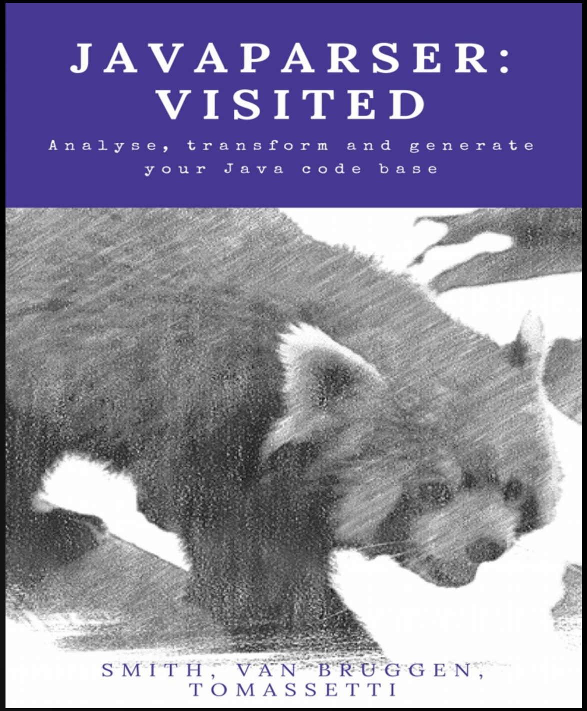
Analyse:
Write code that can traverse Java source and look for the patterns you are interested in.

Transform:
Build tools that can not just identify code patterns, but also has the ability to change them.

Generate:
Be smart, don't spend time writing boiler plate, generate it!

JavaParser

Make sure to get the latest version of JAVAPARSER: VISITED book. This book may be used as a reference while working with Java Parser library.


JAVAPARSER: VISITED
Analyse, transform and generate your Java code base
SMITH, VAN BRUGGEN, TOMASSETTI

Working with JavaParser library using maven project.

1. First thing is to find the Java parser library in Maven Repository. https://mvnrepository.com/artifact/com.github.javaparser/javaparser-core

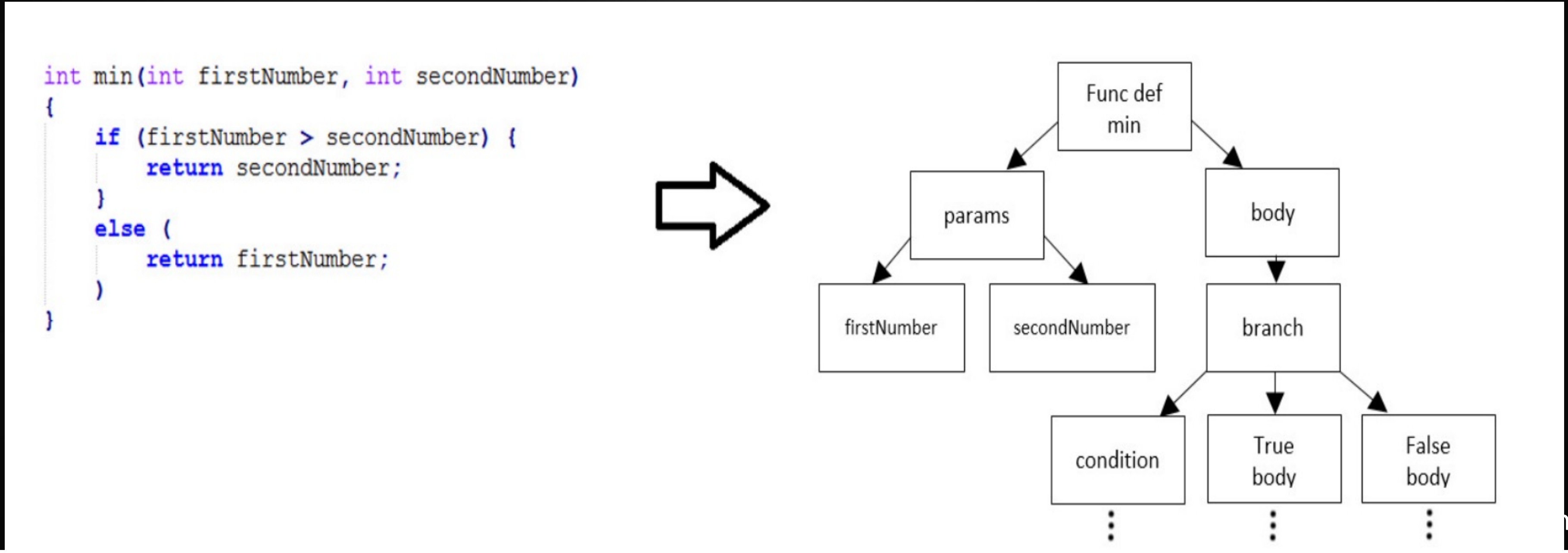
2. Make sure to get the last version and copy the snippet into the pom.xml file of your Maven project.
3. The first step of using the library (simple case) is to create a Compilation Unit.

JavaParser
Setting

1. JavaParser is available as a Java library and we can import it to set up a parser as a Java application to parse other Java projects.
2. The library is available in Maven Repo: https://mvnrepository.com/artifact/com.github.javaparser/javaparser-core
3. Complete the POM.xml.

Abstract Syntax Tree

1. JavaParser converts each Java file into an AST.
    1. AST is a Tree.
    2. Tree is a type of Graph (Network) that has no cycle and there is only and only one path between any two nodes.
2. AST reveals several different pieces of information about the source-code, such as their types, parameters, arguments and a lot more.
3. Each Entity in the source code AST, regardless of their size, is called a Node (or Entity).
4. Each AST has one and only one main Root and may have several (no limit) children and descendants.



```java
int min(int firstNumber, int secondNumber)
{
    if (firstNumber > secondNumber) {
        return secondNumber;
    }
    else (
        return firstNumber;
    )
}
```

JavaParser

Supports all versions of Java

Two desirable aspects of JavaParser:

Component/Entities identification: E.g., methods, variables, classes etc. Containment relations: E.g., parent child.

Relationship/Invocation identification: E.g., method call, variable usages, class inheritance etc.

- ❖ Supports lexical preservation and pretty printing: it means you can parse Java code, modify it and printing it back either with the original formatting or pretty printed.

- ❖ It can be used with JavaSymbolSolver, which gives you symbol resolution. I.e., it understands which methods are invoked, to which declarations references are linked to, it calculates the type of expressions, etc.

Nodes Features

Node and Node-Type:

Each node has a type (Node-Type):

1. Class; e.g., **ClassOrInterfaceDeclaration**
2. Method; e.g., **MethodDeclaration**
3. Variable; e.g., **VariableDeclarator**
4. Etc …

Node relationships:

Nodes can have containment (parent-child) relationship such as:

1. Class A (parent) contains Method B (child)
2. Method C (parent) contains Block Statement (Child)

Nodes can have invocation relationships to each other such as:

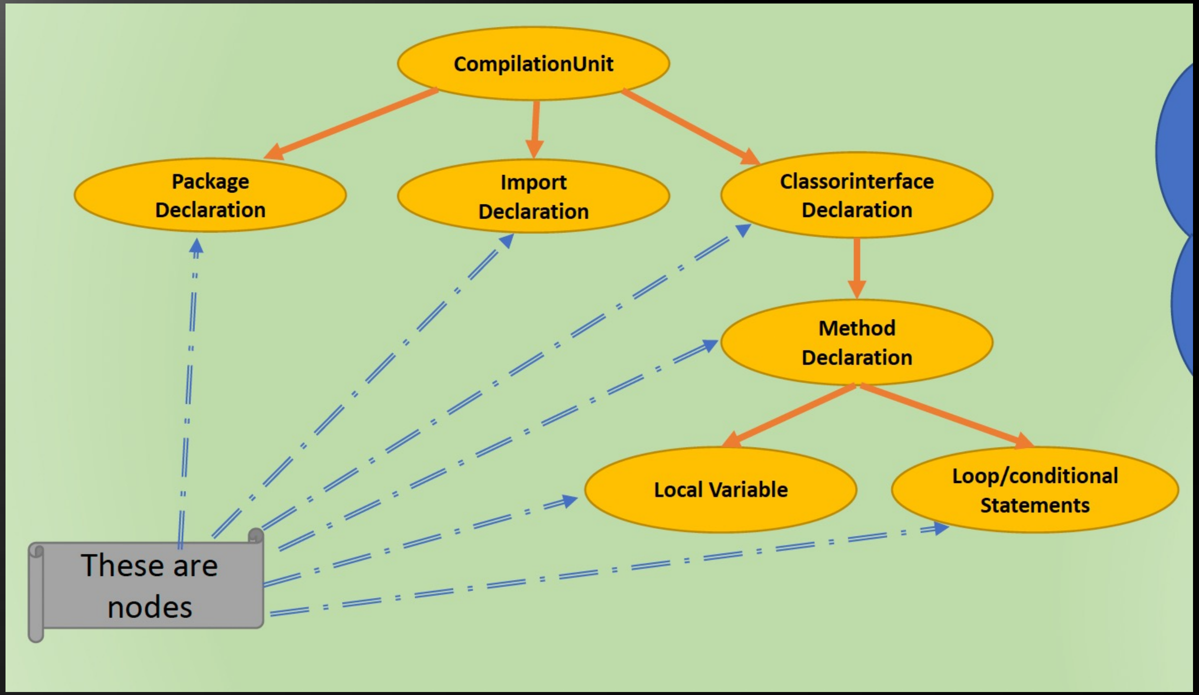
1. Method A calls Method B
2. Class A inherits Class B
3. Method A uses Variable B from a different class.

This type of relationship is reflected in the AST

This is a high level type of relationships that is not initially part of AST.

JavaParser CompilationUnit

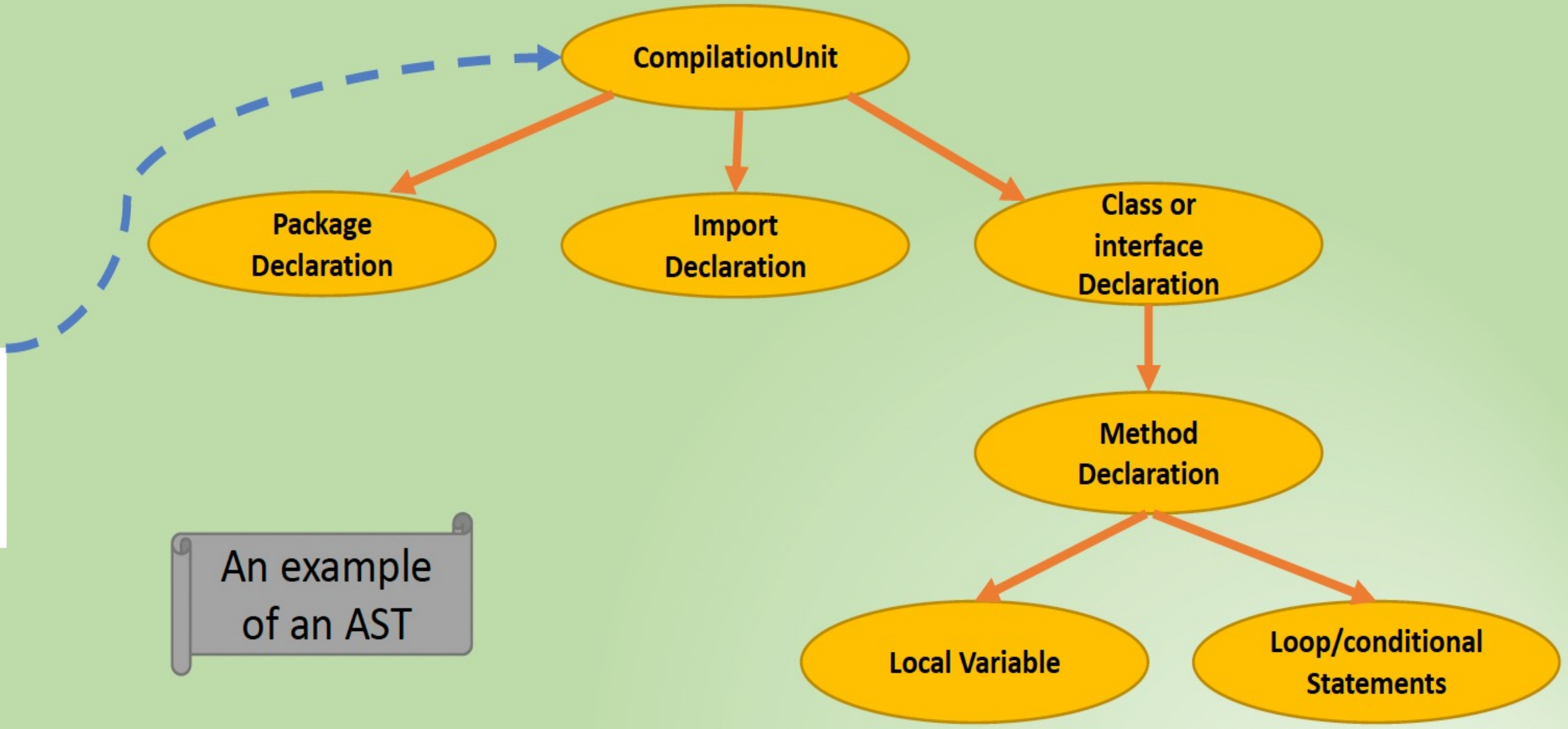
- ***Compilation Unit*** is the main root of each AST. (Note each Java file has one AST).
- The ***Compilation-Unit*** is the Java representation of source code from a complete and syntactically correct java file you have parsed.
- In the context of an AST as mentioned, you can think of the Java file as the main node root.
- From this node, you can have access to all other nodes in the source code e.g., methods, classes, types, variables etc…



JavaParser

How to create the main root of the AST

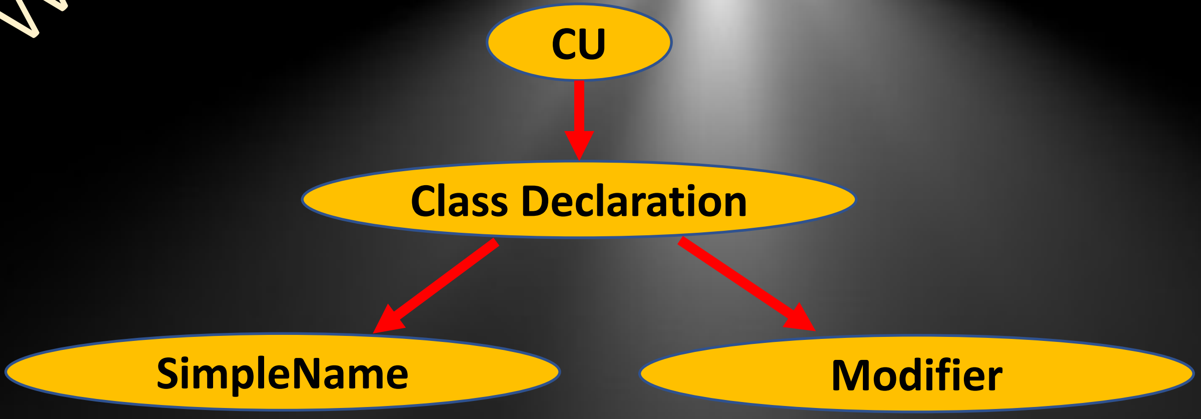
```
File file = new File("myProgram.java");

CompilationUnit compilationUnit =
        StaticJavaParser.parse(file);
```



An example of an AST

Parse a Hello World

```
public class Farshad {

}
```



Note that now, **SimpleName** is the descendent of CU and CU is the ancestor of **SimpleName**. **SimpleName** is the name of the class and not its content.

```
public class Farshad {
}
*****
com.github.javaparser.ast.body.ClassOrInterfaceDeclaration
        public
        *****
        com.github.javaparser.ast.Modifier
        Farshad
        *****
        com.github.javaparser.ast.expr.SimpleName
```

```
public static void main(String[] args) throws FileNotFoundException {


    File file = new File("src/main/resources/test.java");

    CompilationUnit cu =  StaticJavaParser.parse(file);

    for(int i=0;i<cu.getChildNodes().size();i++) {
        System.out.println(cu.getChildNodes().get(i).toString());
        System.out.println("*****");
        System.out.println(cu.getChildNodes().get(i).getClass().getName());


        for(int j=0;j<cu.getChildNodes().get(i).getChildNodes().size();j++) {
            System.out.println("\t"+cu.getChildNodes().get(i).getChildNodes().get(j).toString());
            System.out.println("\t*****");
            System.out.println("\t"+cu.getChildNodes().get(i).getChildNodes().get(j).getClass().getName());
        }
    }
}
```

- The class itself has some other information such as the name of the class and the body (if it has). Therefore the tree can be further extended to the third level to have the children/child of the class declaration.
- Note that every single piece of information in the source code is a node and has a node-Type; e.g., method name, class name, method type/modifier (public or private), code blocks (if, for, while, try catch …)

Node and Node-Type

- ➢ In JavaParser, the generic representation of each Entity is Node.
- ➢ JavaParser initially returns an AST that is comprised of Nodes.
- ➢ Normally the children of each Node are also returned as a set of Nodes.
- ➢ However, each node is responsible for a role in the source code.

- ❑ A Node that represents class ideally should be casted to ClassOrInterfaceDeclaration node-Type.
- ❑ A node that represents method ideally should be casted to MethodDeclaration node-Type.
- ❑ A node that represents class name ideally should be casted to SimpleName node-Type.

- ❖ One of the advantages of casting is to have direct access to all the features of that entity (node-Type) as these features are not directly accessible from Nodes.
- ❖ Another advantage of casting is to collect only a certain Node-Types; e.g., collecting only the class methods and not its fields/variables/methods.
- ❖ Nodes usually have less functions in JavaParser as every single Java Component (e.g., class, method, if block, variable …) is initially made as a node.
- ❖ To get more specific details of each Java component (e.g., Method arguments, class types etc…) nodes may be casted to their Node-Type.

Node and Node-Type

Exercise 2:

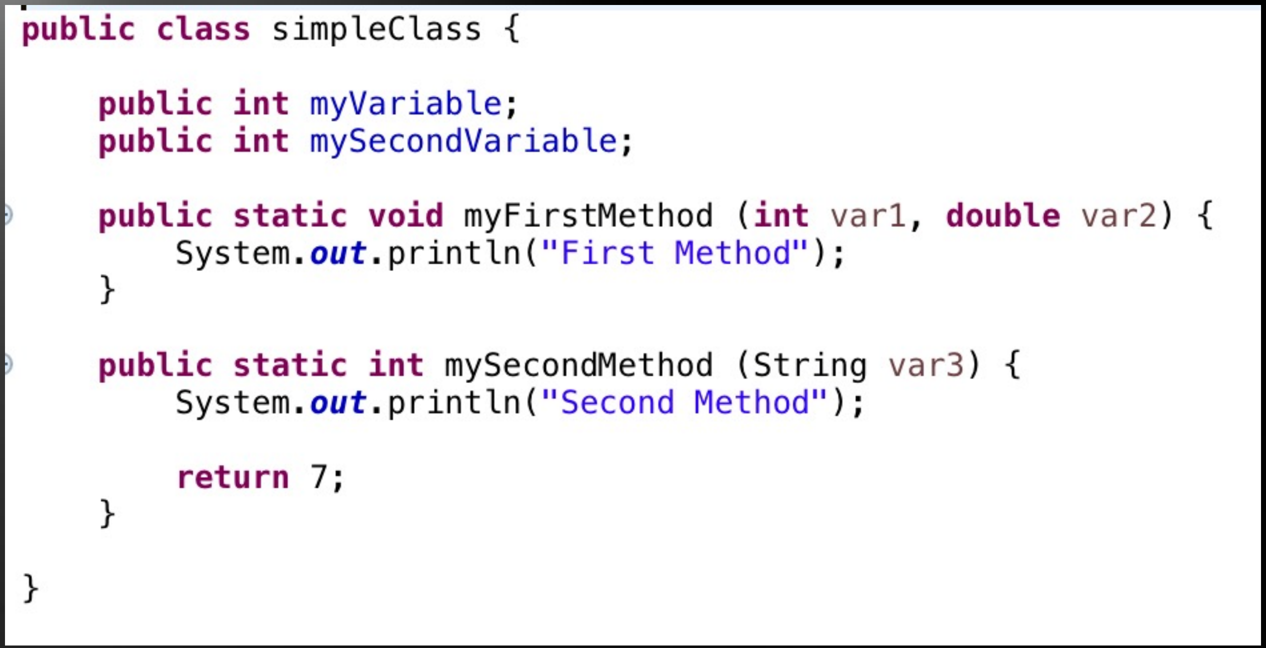
- Parse the following Java class and identify the children and grand children of Compilation Unit.

Exercise 3:

- Parse the following Java class and identify the classes and the methods inside of them only.

Exercise 4:

- Extend the second exercise and get the name of each class and methods as well.

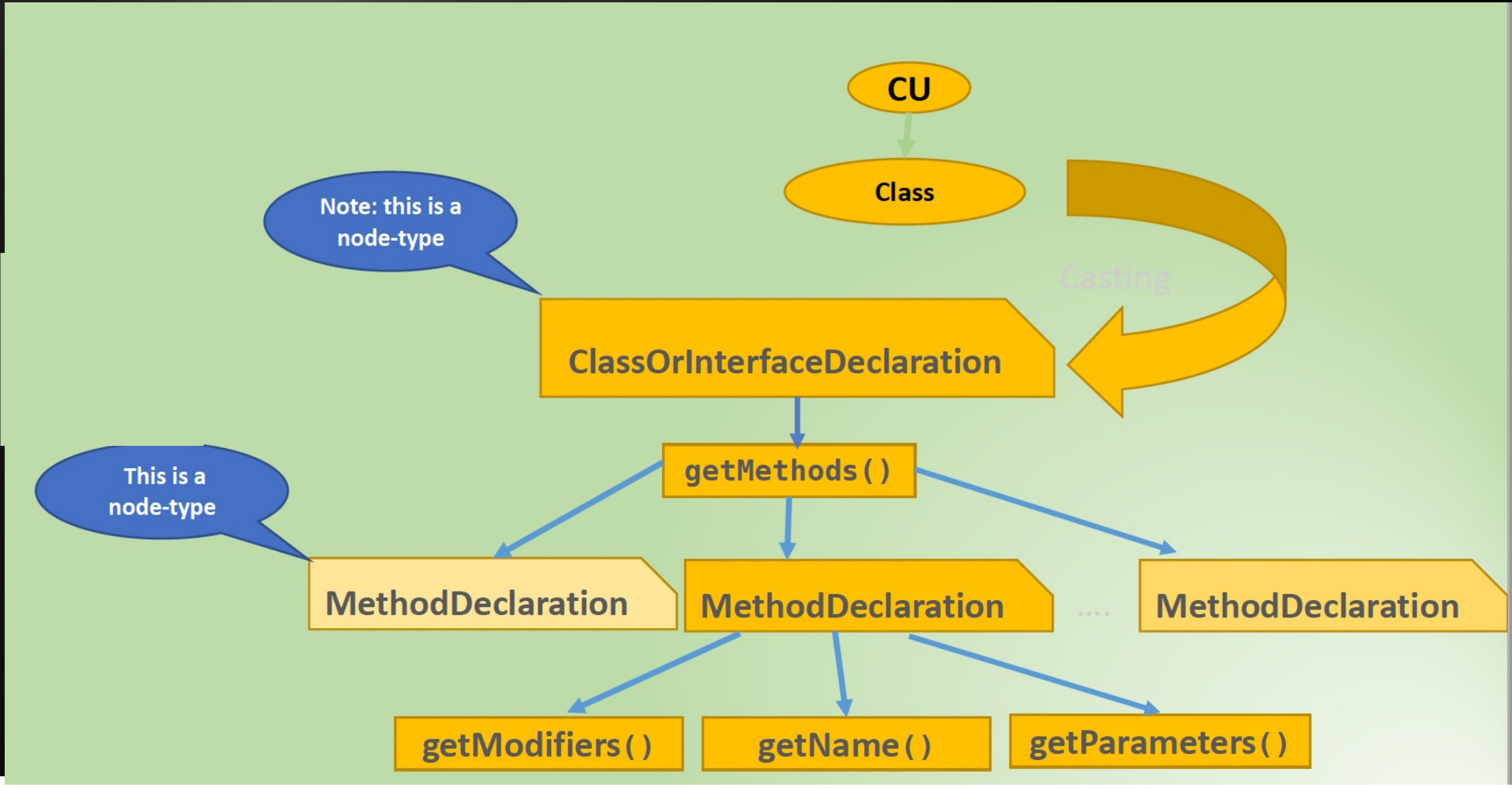


```java
public class simpleClass {

    public int myVariable;
    public int mySecondVariable;

    public static void myFirstMethod (int var1, double var2) {
        System.out.println("First Method");
    }

    public static int mySecondMethod (String var3) {
        System.out.println("Second Method");

        return 7;
    }

}
```

Node Casting

As mentioned earlier, to get the full set of features for each Node, Node should be casted to its actual node-Type. There is more flexibility when Node is casted to its actual node-Type.

Exercise 5:
Parse the Java class from previous slide and identify the methods' names, their parameters, and their modifiers.



Parse the Whole Project

- So far we have seen how we can locally parse a file or a sample java source-code.

- The real-world cases are large projects with several packages and tens of hundreds of Java files, classes and etc.

- Since we do not know how many entities or java files exist in the project, one of the best options is to use recursive functions.

Parse the Whole Project

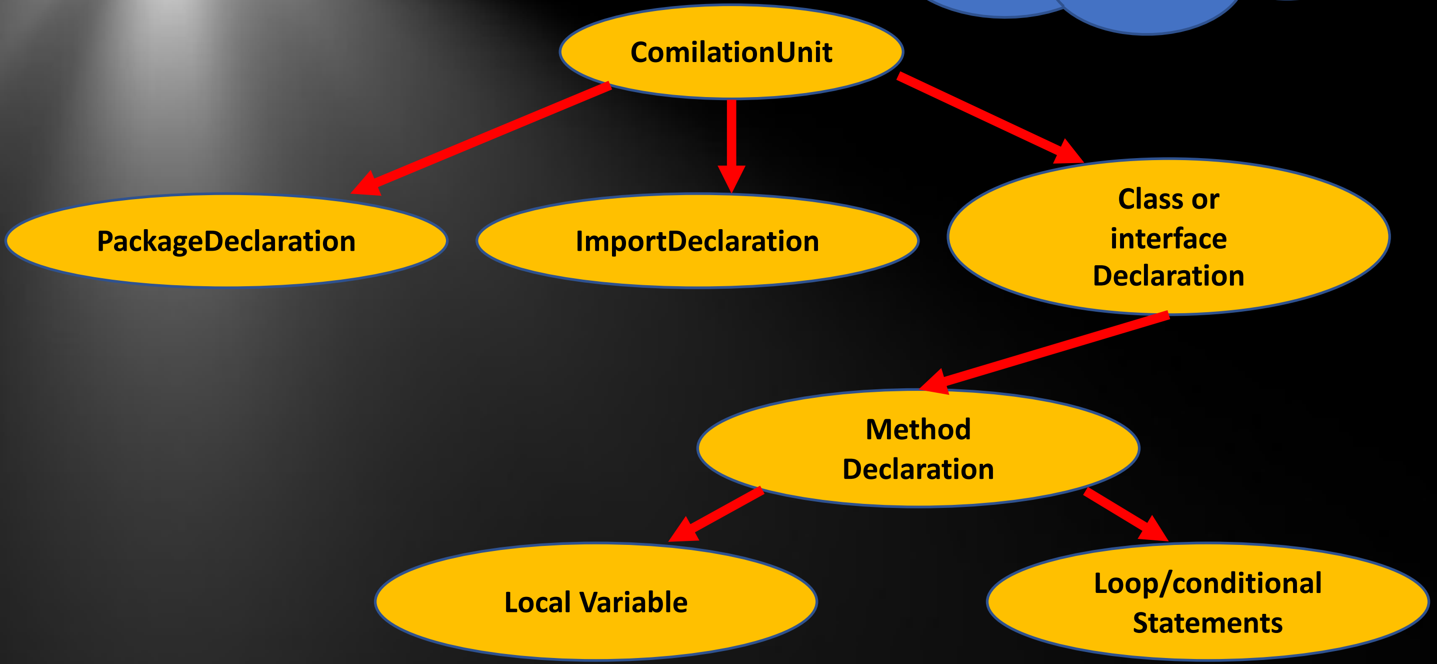
1. From the main project folder, detect all the .java files using Recursive functions. Note that projects might have nested folders therefore all of them must be met recursively.
2. For each .java file create a CU and recursively read all the desired descendants (including children) until the leaves are reached out.

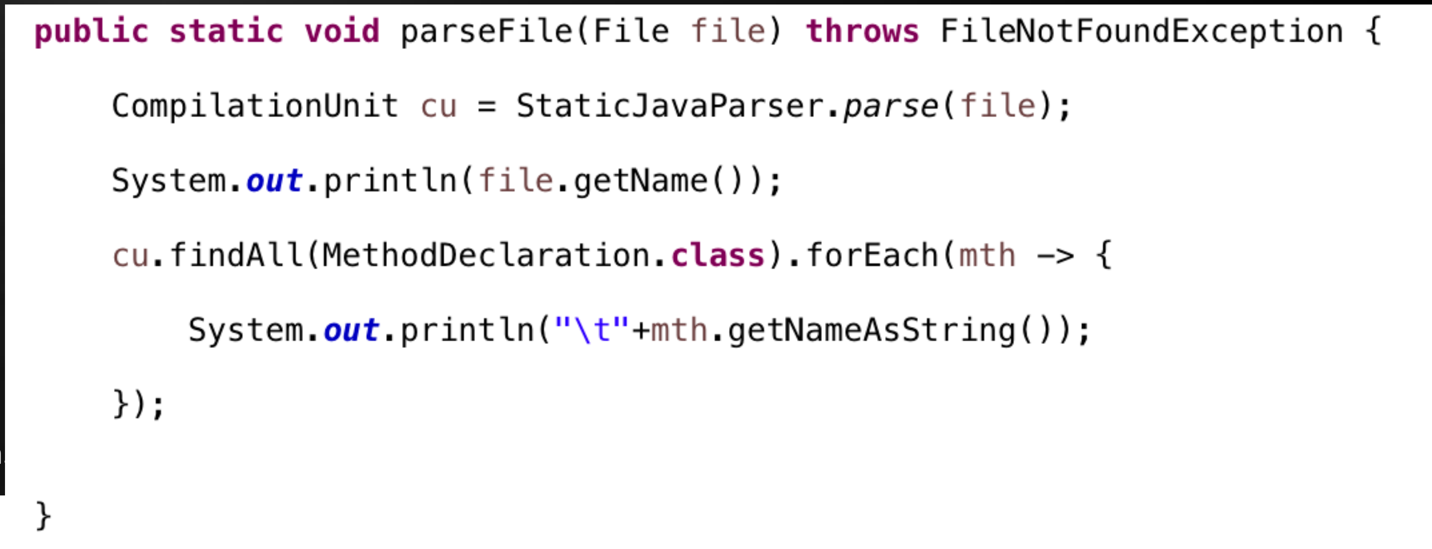
Exercise 6:
Use the junit_test project shared in Canvas and detect only the class names and the java file name they belong to.

Another approach to Travers AST using findAll() method

Look at Appendix B in the JavaParser: Visited book.

1. So far, we were using **getChildNodes**() on a given node (e.g., cu) to get the direct children of the node (e.g., for **CompilationUnit**, we use **getChildNodes**() to get Classes or **PackgeDeclaration**).
2. In order to get the methods, getChildNodes() should be called again on Class nodes and so on.
3. This approach consumes several nested for loops especially if a more granular node is to be detected.
4. JavaParser has another useful function called *findAll*(nodeType) that can be applied on any node in the AST tree (Node.findAll(nodeType).
5. findAll takes a node-Type class (e.g., ClassorInterfaceDeclaration or MethodDeclaration) and traverses the tree rooted at **Node** in pre-order manner and only returns the nodes with the type specified (**nodeType**).




```
public static void parseFile(File file) throws FileNotFoundException {

    CompilationUnit cu = StaticJavaParser.parse(file);

    System.out.println(file.getName());

    cu.findAll(MethodDeclaration.class).forEach(mth -> {

        System.out.println("\t"+mth.getNameAsString());

    });


}
```

Decedents Nods

Exercise 7:

Use the junit_test project shared in Canvas and detect all the methods and their parents (the class they belong to).

How to deal with Variables

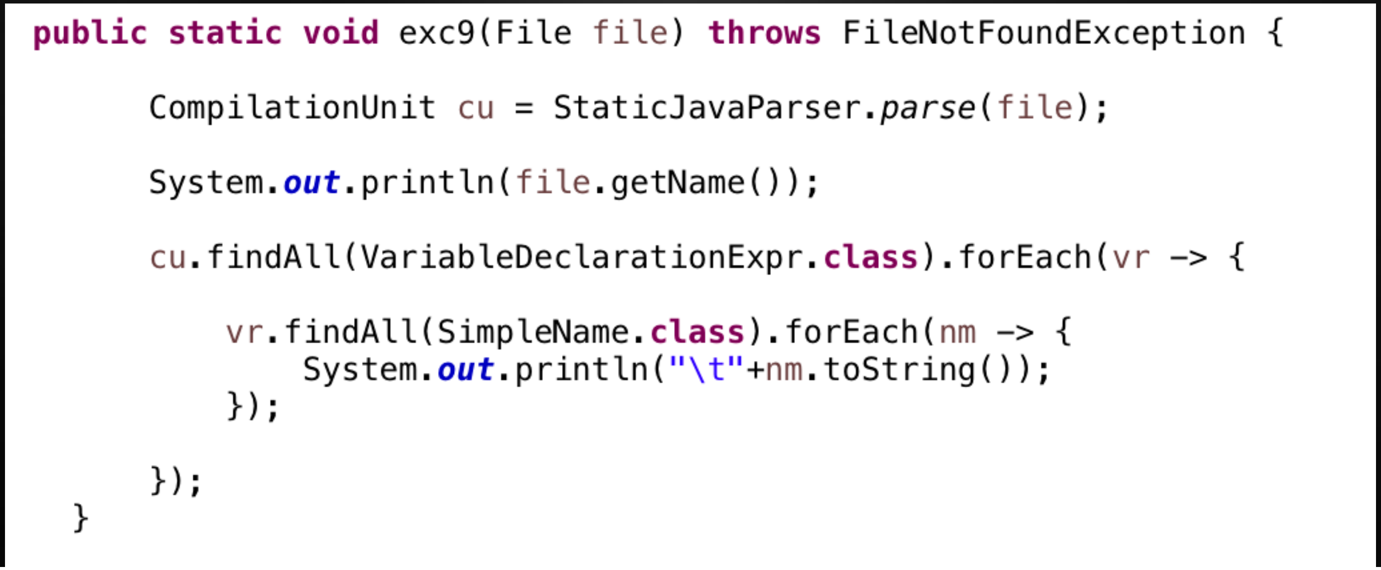
- JavaParser has sophisticated mechanism to deal with both local and global variables.
- Variables that are declared as a class field (static or class field) are dealt with: **FieldDeclaration**
- Variables that are declared locally inside of a method are dealt with: **VariableDeclarationExpr**
- Each casted node (either **FieldDeclaration** or **VariableDeclarationExpr**) has a number of methods that can extract more detailed information about the variable, e.g., modifier, type, variable name and etc.

Exercise 8:
Use the junit_test project shared in Canvas and detect all the local variables (i.e., the variables that are declared inside of the methods).

How to deal with Variables

❖ **In order to find the names of all variables one could retrieve the SimpleName nodes of FieldDeclaration** and **VariableDeclarationExpr.**

Exercise 9:
Use the junit_test project shared in Canvas and detect all the names of local variables (i.e., the variables that are declared inside of the methods).



```
public static void exc9(File file) throws FileNotFoundException {

        CompilationUnit cu = StaticJavaParser.parse(file);

        System.out.println(file.getName());

        cu.findAll(VariableDeclarationExpr.class).forEach(vr -> {

            vr.findAll(SimpleName.class).forEach(nm -> {
                System.out.println("\t"+nm.toString());
            });

        });
    }
```

Another Approach to travers AST Visitors

- JavaParser has a an abstract class called: VoidVisitor
- List of the methods of this class can be found at:

https://www.javadoc.io/doc/com.github.javaparser/javaparsercore/2.2.1/com/github/javaparser/ast/visitor/VoidVisitorAdapter.html

- We can create a class that extends VoidVisitorAdapter class.
- Depending on what we are looking for, we can override the method and get the desired results.
- The list of all Visitable Nodes (components of a Java source code) is found in the Appendix B of the JavaParser: Visited Book at: https://leanpub.com/javaparservisited

Exercise 10:
Use the junit_test project shared in Canvas and detect all the methods names along with their class name using Visitors.

Visitors

Exercise 11:
Use the junit_test project shared in Canvas and detect all the class with more than 2 methods. Print the name of the class along with the number of methods and fields they contain.

```java
static class Exc11_ClassIdentification  extends  VoidVisitorAdapter<Void> {

    public void visit(ClassOrInterfaceDeclaration cls, Void arg)  {
        super.visit(cls, arg);
        if(cls.getMethods().size()>2) {
            System.out.println(cls.getNameAsString()+" class with "+cls.getFields().size()+" number of parameters has "
                    +cls.getMethods().size()+" number of methods");
        }
    }
}
```

Node Accessibility in JavaParser

Nodes Children	Nodes Descendants	Visitors
It starts from CU and goes level by level. It might require several for loops until the desired node-Type is reached out.	A given node can be used to find all its descendants with a particular type. No extra loop is needed. Pre-order traverse is used to go through the whole rooted tree.	It can start from CU. Every node type has a particular function that can be customized. It uses Visitor Design Pattern.
node.getChildNodes()	node.findAll(SimpleName.class)	super.visit(mth, arg);

Use Cases

- ❑ Most of the tasks and exercises in this module require a Java project. Github can be used to search and find OPEN-SOURCE and FREE projects.
- ❑ It is recommended to work and practice with relatively large projects.
- ❑ JHotDraw is a graphics (two-dimensional) framework for structured drawing editors that is written in Java. It is based on Erich Gamma's JHotDraw, which is copyright 1996, 1997 by IFA Informatik and Erich Gamma.
- ❑ JHotDraw as an example is provided for you and is shared in Canvas.

How to represent code analysis findings

The finding of code analysis may be represented in different ways.

One of the methods to represent the code analytical finding is to visualize the findings (e.g., chart, diagram, etc.).

How to represent code analysis findings

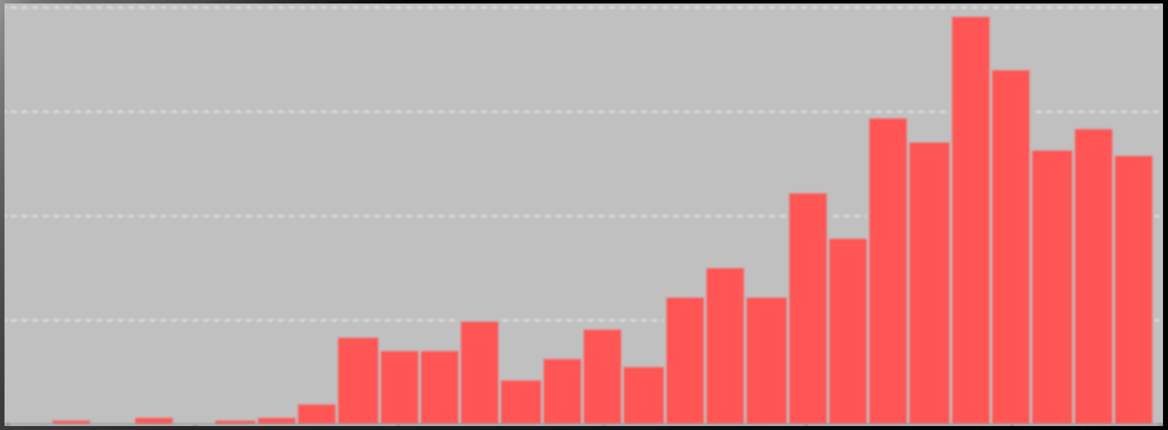
JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications.

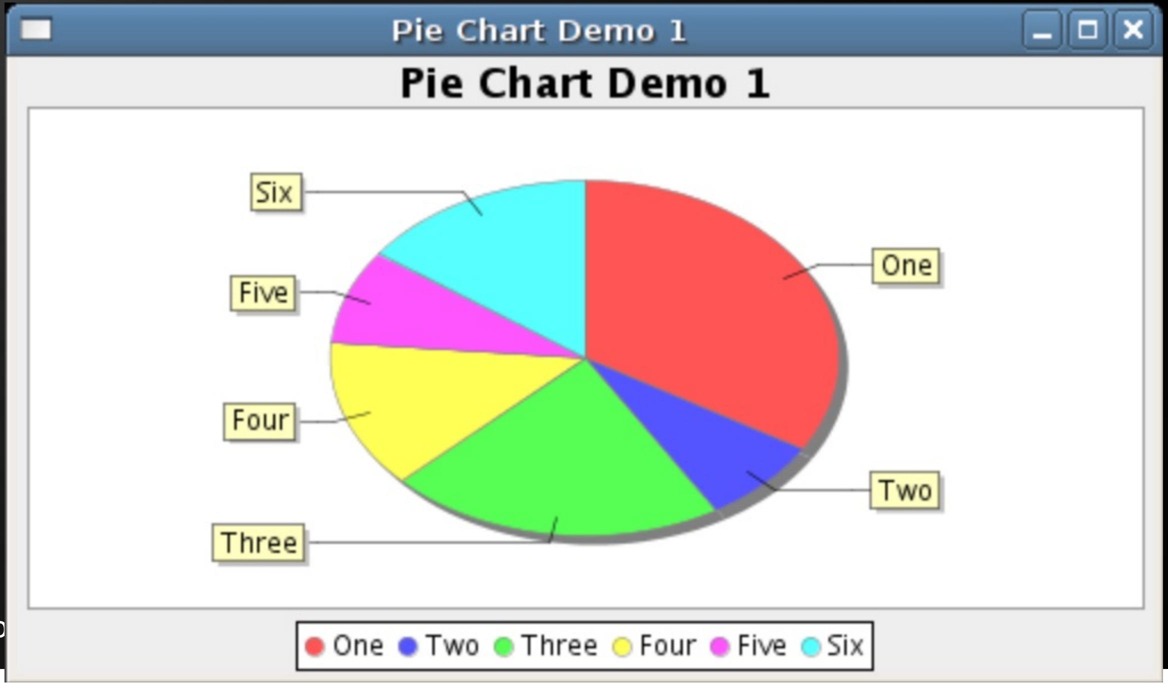
How to represent code analysis findings

- ❑ Bar chart.
- ❑ Pie chart.
- ❑ Histogram.
- ❑ Scatter plot.
- ❑ and many more ..

JFreeChart is available in Maven Repository:

https://mvnrepository.com/artifact/org.jfree/jfreechart





Thank you