Program Parser

Relations in the Source-Code

Munster Technological University

Dr Farshad Ghassemi Toosi

Review

- Parent child relationship (containment relationship).
- Visitors.
- FindAll(Type.class) : it finds all descendent nodes with a given Type
- findAncestor(Type.class) : it finds all ancestor nodes with a given Type.
- ChildNode
- ParentNode
- Types and modifiers…

- Exercise 0:

Use Javaparser and detect all methods with some of their features…
Use JHotDraw project.

How programmatically Parse a Programming Language

Books are essentially sequential, so if you make a change on page 126, knock-on effects can only happen from page 126 on. But software is not sequential in nature so a change on 'page 126' can potentially impact anywhere in the system.

(Jim Buckley, Why maintaining software is like updating War & Peace)

https://www.rte.ie/brainstorm/2021/0203/1194753-software-coding-maintenance-updating-war-and-peace/

There are several different types of relations between components/nodes in the source code. These relations most of the times create dependencies in the source-code (see this link).

Some of the dependency types are listed below:

1) Class Dependencies
2) Interface Dependencies
3) Method / Field Dependencies

Source-code dependency is **BAD** but inevitable.

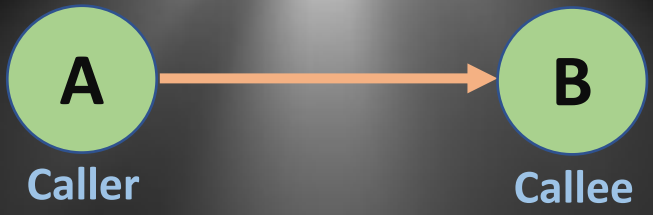
Why source-code dependency is bad?

"Dependencies are bad because they decrease reuse. Decreased reuse is bad for many reasons. Often reuse has positive impact on development speed, code quality, code readability etc."

Relationship in Source-Code

- The dependencies or relations between components may be strong (tight) or weak (loose).
- Usually a tight dependency between components make the future updates a bit more difficult. Why?
- As part of the source-code analysis, it might be beneficial to identify these dependencies.
- JavaParser has sophisticated and strong approaches to easily identify these relations.

Relationship Detection

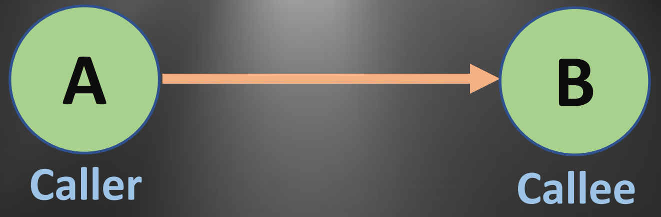


Every relationship has two ends Caller and Callee.

Caller and Callee, depending on the programming language, may have different types:

1) **A** and **B** can be two methods/functions.
2) **A** and **B** can be two classes one (**A**) inherits the other one (**B**).
3) **A** can be a class and **B** can be a field that is accessed by **A**.
4) …

Relationship Detection



Every relationship has two ends Caller and Callee.

JavaParser has different mechanism to detect different types relationships.

- ❑ ***Shallow Detection:*** *The relationship is detected but only the details of the Caller is identified, the details of the Callee is still not fully clear.*
- ❑ ***Deep Detection:*** *The relationship is detected and the details of both Caller and Callee are identified.*

Method/Function calls (Relationship) Detection
Shallow-Detection

- ❑ AST in JavaParser has another Node-Type called MethodCallExpr
- ❑ MethodCallExpr detects the method calls in the source-code.
- ❑ It also detects method calls to the Java libraries methods; e.g., system.out.println("a");
- ❑ The source code does not have to compile.
- ❑ The detected calls are not expressive enough; (e.g., no information about where the **callee** method comes from or what is the type of the **callee** method, just the name of the **callee** can be detected).
- ❑ In order to find method calls, MethodCallExpr that are the descendants of MethodDeclaration must be identified.

Exercise 1:

- o Use the JavaParser and detect all the method calls from the junit-tests-master project.

Variable usage (Relationship) Detection
Shallow-Detection

- ❑ AST in JavaParser has another Node-Type called FieldAccessExpr.
- ❑ FieldAccessExpr detects all the places in the class that a field is accessed whether from the same class or other classes.
- ❑ The source code does not have to compile.
- ❑ The detected FieldAccessExpr are not expressive enough; e.g., no detailed information about where the field was declared.
- ❑ In order to find variable usages, FieldAccessExpr that are the descendants of ClassOrInterfaceDeclaration must be identified.

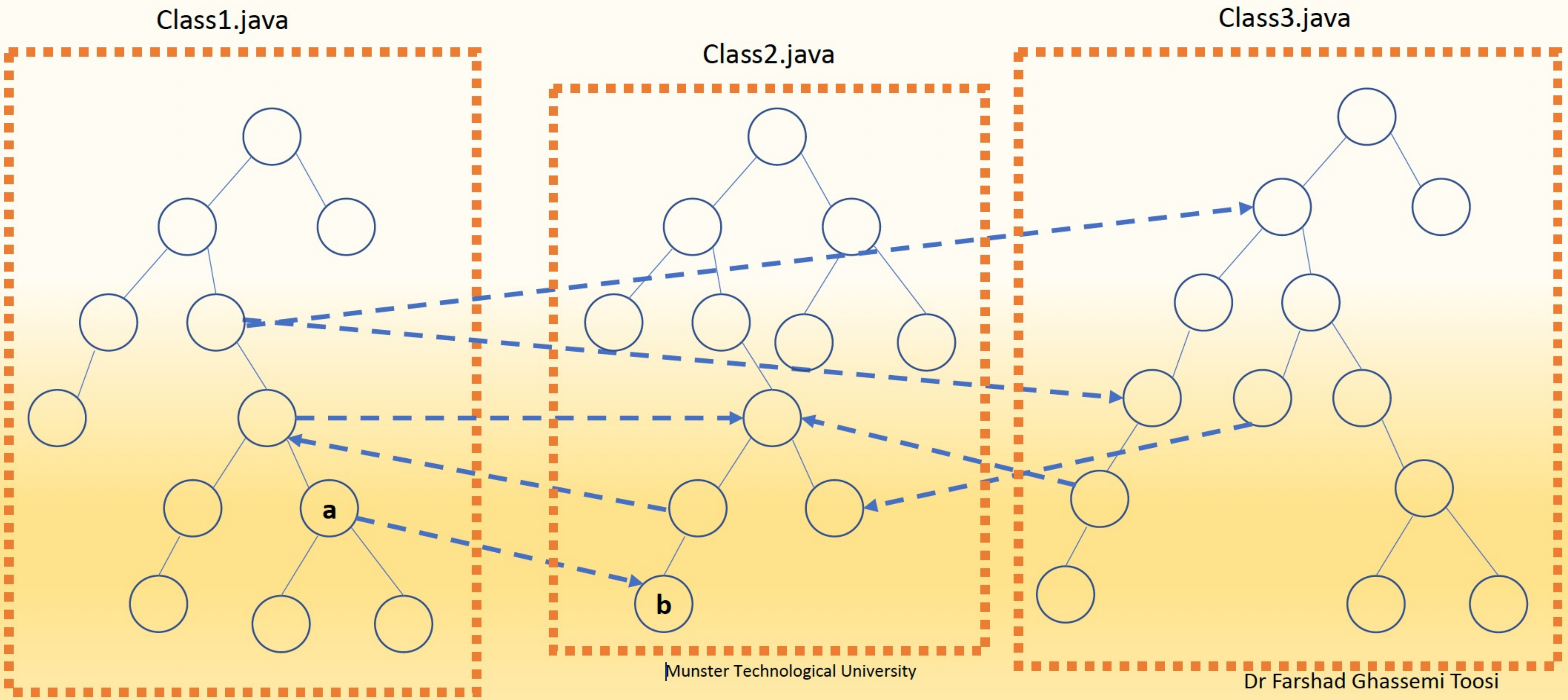
Exercise 2:

Use the JavaParser and detect all field accesses in the junit-tests-master project.

Relations Identification using JavaParser.

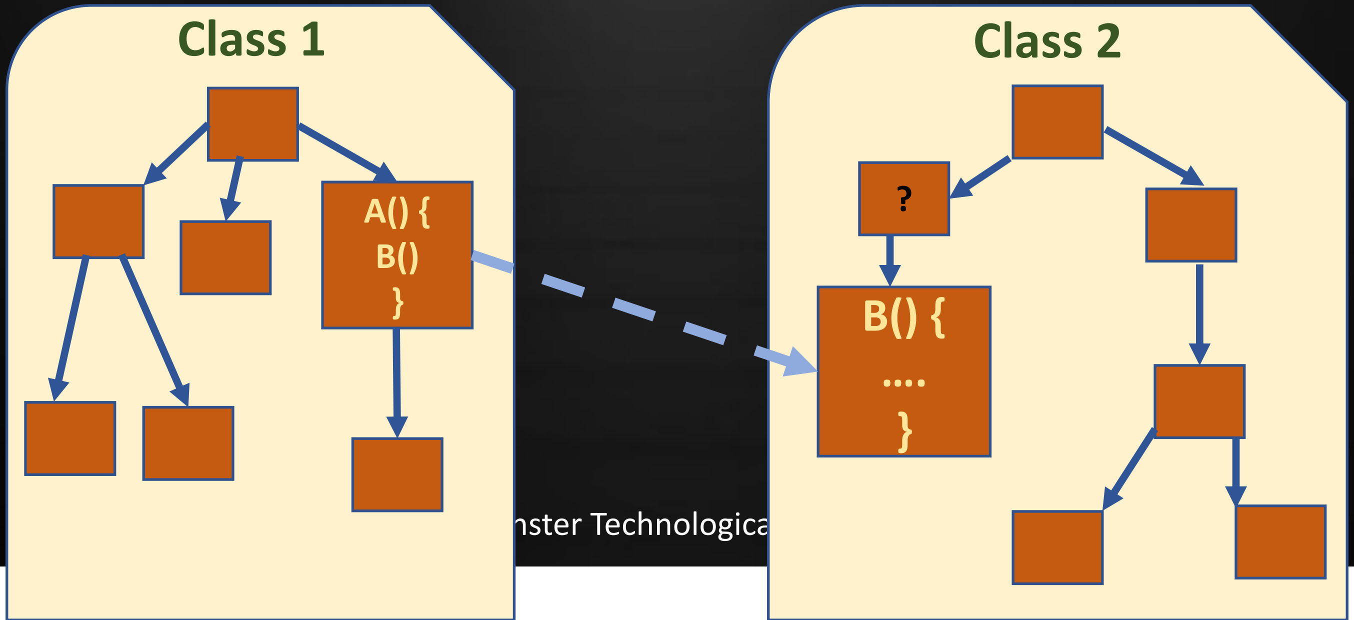
- ❑ JavaParser, like other parsers, takes information present in source code and organizes it into a tree, (i.e., Abstract Syntax Tree).
- ❑ In many situations, the information present in the AST is sufficient for the user’s needs.
- ❑ In other cases however you may need to elaborate the AST and calculate additional information. In particular, you may want to resolve references and find the details of a relation between nodes (specially the details of Callee).
- ❑ In a sense, this means tracing new links and transforming a tree into a graph (see next slide).

Relations between multiple ASTs

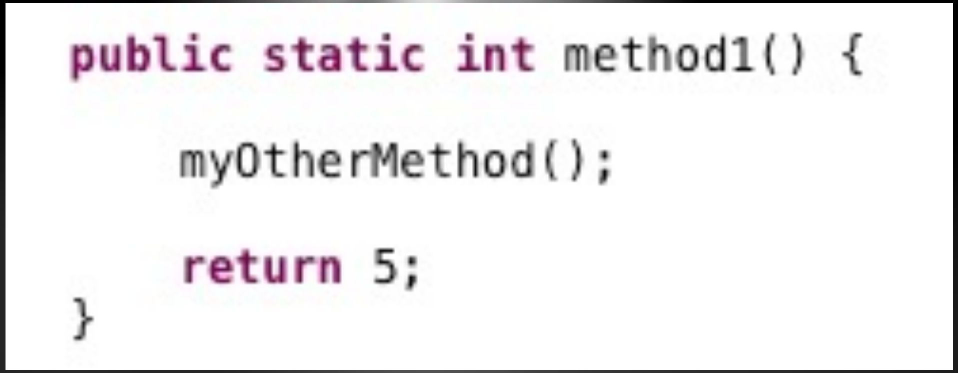


Relations Identification using JavaParser.

- JavaParser, parses each java file independent to the other java files.
- When parsing ***class 1***, the relation between ***A (in class 1)*** to ***B (in class 2)*** is identified but the details is limited to what is existing in ***class 1***; e.g., the parent node of ***B*** in ***class 2*** is unknown. There is no information on what is happing in ***B***.
- This is because, JavaParser is parsing ***Class 1*** at the moment.
- Once JavaParser starts parsing ***class 2***, the details of ***class 1*** are forgotten, and the focus is moved onto ***class 2*** now.



JavaParser and Method calls



```
public static int method1() {

    myOtherMethod();

    return 5;
}
```

- As mentioned in previous slides, JavaParser still can parse a source-code even if it misses a reference declaration.
- In this example, ***method1()*** calls ***myOtherMethod()*** but the declaration of ***myOtherMethod()*** does not have to exist in order to parse the current class.
- JavaParser still detects this method invocation: ***method1()*** ***myotherMethod()*** even if the declaration of ***myOtherMethod()*** does not exist in the current java file.
- The detected method-invocations by JavaParser is a shallow-detection, therefore, there is no detail about the **Callee** method, e.g., the type of the method, return type, contained class (parent class), its declaration and etc.
- This type of method invocation is detected by MethodCallExpr class in JavaParser as seen in previous slides/examples.

Reference Solver

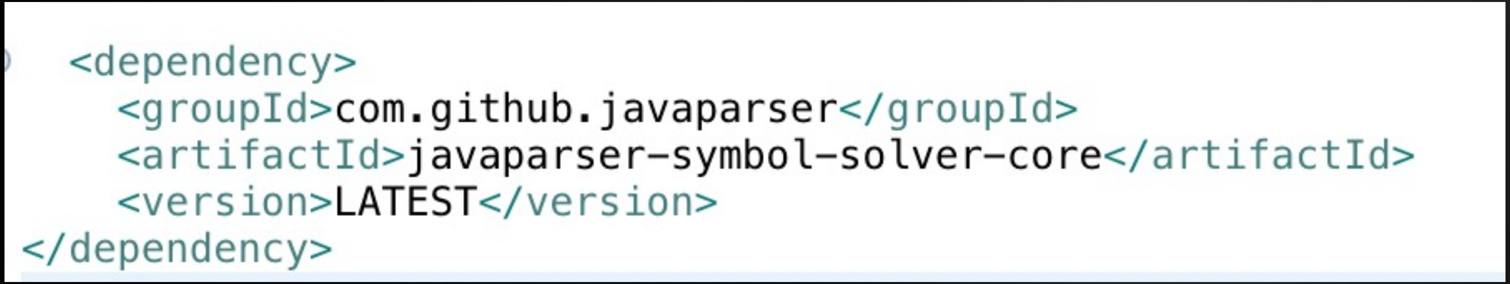
- The solution to this issue (shallow detection) is to use another library called javasymbol-solver.
- The symbol solver was born as a separate project to be used with JavaParser without changing the JavaParser API.
- The java-symbol-solver permits you to resolve symbols in Java source code.
- It is intended to be used with JavaParser. After JavaParser has produced an Abstract Syntax Tree (AST) for a source file, java-symbol-solver can be used to do things like:
  - Find out to which definition a certain value refers to,
  - Calculate the type of a certain expression,
  - Details of a destination node in a relation (e.g., callee method etc).

JavaParser will not answer the question "**where the callee method/variable is defined?**" You will need another library for that, for which we recommend the JavaSymbolSolver.

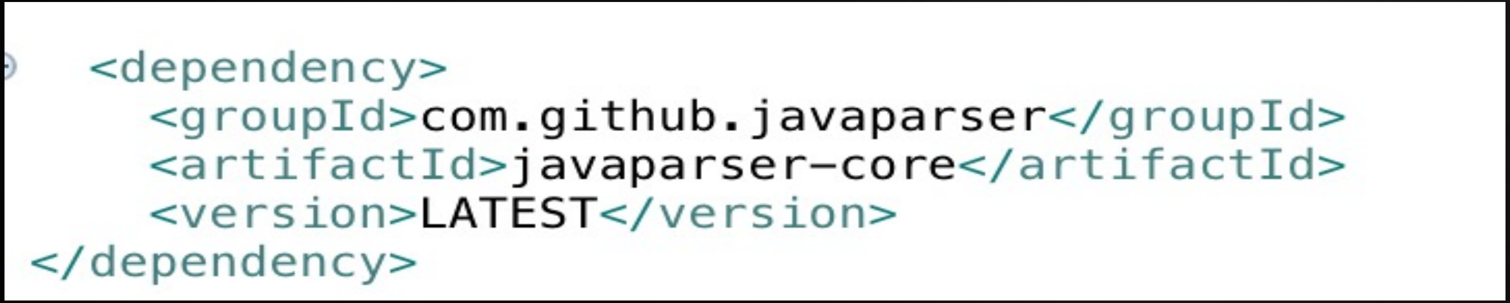
Reference Solver Setup

- ❑ In order to use Symbol solver, another library needs to be imported into the project.
- ❑ The library is available in Maven repository.

https://mvnrepository.com/artifact/com.github.javaparser/javaparser-symbol-solver-core/3.24.0



```
  <dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-symbol-solver-core</artifactId>
    <version>LATEST</version>
</dependency>
```



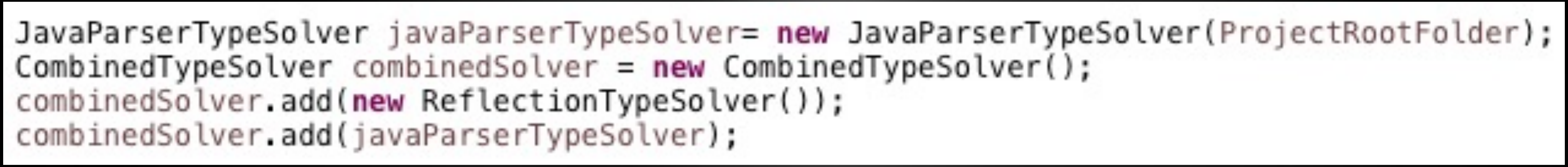
```
  <dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-core</artifactId>
    <version>LATEST</version>
</dependency>
```

Change the version to "LATEST".

How to make a Simple Symbol Solver Program

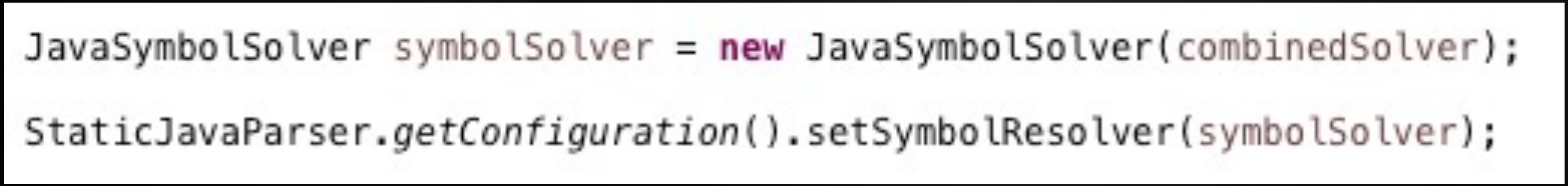
In order to apply symbol solver:

1. The entire project needs to be introduced to the symbol solver



```
JavaParserTypeSolver javaParserTypeSolver= new JavaParserTypeSolver(ProjectRootFolder);
CombinedTypeSolver combinedSolver = new CombinedTypeSolver();
combinedSolver.add(new ReflectionTypeSolver());
combinedSolver.add(javaParserTypeSolver);
```

2. Bonding the symbol solver with the JavaParser:



```
JavaSymbolSolver symbolSolver = new JavaSymbolSolver(combinedSolver);

StaticJavaParser.getConfiguration().setSymbolResolver(symbolSolver);
```

What is a **Type Solver**? It is the object which knows where to look for classes. When processing source code you will typically have references to code that is not yet compiled, but it is just present in other source files.

Once the symbol-Solver is properly set and bonded with JavaParser, each node may be resolved, (e.g., MethodCallExpr) to get the details of the callee nodes.

Exercise 3:

Use the JavaParser and SymbolSolver and detect all method calls and their types. Use junit-tests-master project.

```
File project = new File("src/main/resources/junit-tests-master/src/main/java");
```

Note, when working with symbol solver the project should be addressed more precisely at thepackage level only.

Symbol Solver Advantages

- ❑ One of the advantages of symbol solver is when there are overloaded methods.
- ❑ Overloaded methods in Java are those methods that have the same name (or even sometimes the same input arguments) but different return type.
- ❑ The shallow call detection wouldn’t be able to distinguish them but Symbol Solver can.

Symbol Solver Advantages

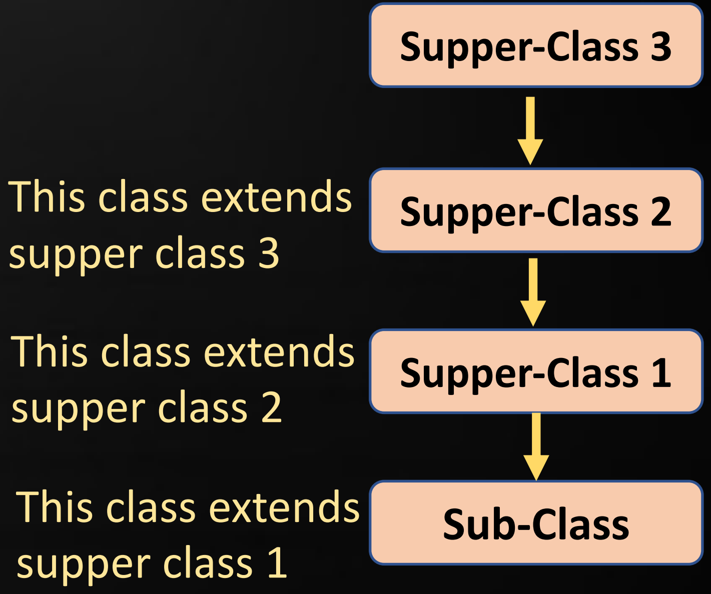
- Another advantage of using Symbol Solver is to track the class fields from ***Class A*** that are accessed by ***Class B***.
- FieldAccessExpr is a node-type that detects all the class fields that are accessed in the current class.
- The shallow detection only detects the usages of a class fields, however more details (e.g., where the field class was declared, the type of the field class) can only be found by making use of Symbol Solver.
- Note: The list of all JavaParser nodes are available at the end of JAVAPARSER: VISTED book as Appendix B.

Class Inheritance

- ❑ Class inheritance is one of the Object Oriented principles.
- ❑ Sometimes it is important to find out to what extent, Class inheritance paradigm has been used.
- ❑ The class inheritance discovery is usually made using Symbol Solver.
- ❑ Class inheritance is a link between two classes, therefore, the symbols should be resolved first.

Class Inheritance using Symbol Solver

1) Resolve each Class (Not an interface) to a ResolvedClassDeclaration
2) Call *.getAllSuperClasses()* to find the super-class of the given class.
3) Note that the Symbol Solver needs to be first adjusted (as before).
   1) Sometimes a class have multiple superclass with different levels.
   2) For instance, the sub class inherits super class 1
   3) while super class 1 inherits super class 2 and
   4) Super class 2 inherits super class 3.


Supper-Class 3
This class extends supper class 3
Supper-Class 2
This class extends supper class 2
Supper-Class 1
This class extends supper class 1
Sub-Class

Exercise

- ❑ Exercise 4:

Use the JavaParser and SymbolSolver and detect all the class inheritances. Use JHotDraw project.

Static Class

- ❑ Classes can be defined within other classes and they can be locally used.

- ❑ In order to detect an inner class, you can detect ClassOrInterfaceDeclaration Node-Types that is a child of another ClassOrInterfaceDeclaration Node-Type.

Class Inheritance…

- ❑ Class inheritance does not always happen from classes in the subject system.
- ❑ Class inheritance can be between a class in the subject system (sub class) and a class from an imported library or a native library (super class).
- ❑ Class inheritance creates dependency or coupling between classes.
- ❑ JavaParser and Symbol-Solver both together are able to differentiate these type of inheritance.

Exercise 5:
Repat Exercise 4 and this time only detect class inheritance from the source-code. Class inheritance to outside of the subject project (libraries) should be ignored. Use the JavaParser and SymbolSolver and detect all the class inheritances. Use JHotDraw project as the subject system.