JAVAPARSER: VISITED

Analyse, transform and generate your Java code base

SMITH, VAN BRUGGEN, TOMASSETTI

JavaParser: Visited

Analyse, transform and generate your Java code base

Nicholas Smith, Danny van Bruggen and Federico Tomassetti

This book is for sale at http://leanpub.com/javaparservisited

This version was published on 2023-07-21

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2023 Nicholas Smith, Danny van Bruggen and Federico Tomassetti

Tweet This Book!

Please help Nicholas Smith, Danny van Bruggen and Federico Tomassetti by spreading the word about this book on Twitter!

The suggested hashtag for this book is #JavaParser.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#JavaParser

Contents

Preface 1
- What is JavaParser? 1
- JavaParser is not 1
- Who is this book for? 1
- This book is not for 2
- How to use this book? 2
- Conventions used in this book 3
- Code Samples 3
- How to Contact Us 3
- The Authors 4
- Acknowledgements 4

A Brief Introduction to Abstract Syntax Trees 5
- Everything is a node 5
- When is a node, not a node? 6
- Growing our first tree 6
- Beyond Abstract 8

A Flying Visit 9
- Travelling Companions 9
- A Simple Visitor 10
- A Simple Visitor With State 13
- A Simple Modifying Visitor 15
- A Simple Comment Reporter 18

Comments - Here Be Dragons 22
- Comment Attribution 23
- Comments In Practice 26

Pretty Printing and Lexical Preservation 30
- What is Pretty-Printing? 30
- What is Lexical-Preserving Printing? 31
- Choosing between Pretty Printing and Lexical-Preserving Printing 31
- Using Pretty Printing 32
- Using Lexical-Preserving Printing 34

How it works: the NodeText and the Concrete Syntax Model 37
Summary 40

Solving Symbols and References 41
How to setup the JavaParser Symbol Solver? 42
How to get the type of references: a first example 43
Specifying where to look for types 44
Resolving a type using an absolute name 45
Resolving a Type in a context 47
Resolving method calls 49
Using the CombinedTypeSolver 50
Using the MemoryTypeSolver 51
Summary 52

Appendix A - ReversePolishNotation.java 53

Appendix B - Visitable Nodes 56

Preface

What is JavaParser?

In its simplest form, the JavaParser library allows you to interact with Java source code as a Java object representation in a Java environment. More formally we refer to this object representation as an Abstract Syntax Tree (AST).

Additionally, it provides a convenience mechanism to navigate the tree with what we have termed as *Visitor Support*. This gives developers the ability to focus on identifying interesting patterns in their source, rather than writing laborious tree traversal code.

The final principal feature of the library is the ability to manipulate the underlying structure of the source code. This can then be written to a file, providing developers with the facility to build their own code generating software.

JavaParser is not

Although the library lends itself being used as part of one, it is not a code refactoring toolkit. This is perhaps the most popular misconception about the library we see.

Think of the library as providing a mechanism to answer the question "**what** is this code?", the **why** and the **how** you might choose to manipulate or report on it is up to you, the language engineer.

It is also not a compiler. Syntactically correct source code that can be parsed by the library does not necessarily mean it will successfully compile. Although files that successfully compile are inherently syntactically correct, parsing is just one stage of the compilation process. For example, if you refer to the variable `v` without having defined it, that is syntactically correct, but will lead to a semantic error in the compilation process.

Who is this book for?

Without sounding too facetious: most Java developers. If not the JavaParser library, they will likely benefit from understanding what parsers can offer with regards to their day to day work.

To labour the point, in the majority of cases when choosing to parse source code you are looking either to identify something, usually issues in the code, or ways to automated the generation of code. Both of these practices are largely considered with attaining quality and efficiency improvements;

can I automatically detect problems, can I produce this code more quickly? All developers should be interested in how they can attain both of these.

In likelihood, however, you will rarely be including the JavaParser library in end user software. Typically it forms part of a tooling component that will operate on the source code of your project, or perhaps generate the code that will ultimately service users.

Language tooling is probably the most common example of what we see the library used for. This notion that we can identify points of interest within the code and then perform an action.

More broadly speaking it is for language engineers and tooling developers.

As a library for Java, it is assumed you already have a fair understanding of Java.

This book is not for

Those wishing to learn Java, by extension of the above this book is not going to teach readers Java.

It will not give you great insight into how to write a parser either, beyond what features users will be interested in.

We are also not going to take a deep look into the Java grammar or the underlying grammar definition of the library. Those readers wish to gain more understanding in that area should visit the Java Language Specification[1] site.

Perhaps most controversially, those who are squeamish about lack of tests. Like most things, approaches to testing is subjective and it is not a goal of this book to offer opinions on the subject. As such the example is in this book are largely demonstrated through the `main` method.

How to use this book?

This book is intended to be used as a learning aid for those using the JavaParser library.

It is not a reference guide, we have tried our best to maintain the JavaDoc[2] for this purpose. If you find that documentation lacking please raise a question on GitHub.

We have separated the book into two parts, the first is intended to get you up to speed with Abstract Syntax Trees, the key components of the library and how to get some working examples on your machine. If you are a novice, it is recommended that you read the first part in order to gain a good grounding in the subject.

The second part of the book is a collection of use cases that provide a more detailed look at how we often see the library being used. The use cases there can be read out of order, perhaps starting with one the user relates to most. Although there is certainly no harm in reading them all.

[1]https://docs.oracle.com/javase/specs/jls/se8/html/
[2]http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/

Conventions used in this book

Italic - Used in the first introduction of an important concept within the subject matter.

Bold - Emphasis a particular concept within the current subject context.

`Code` - Used to highlight programmatic code words inline

Code Samples

You are free to use all the code samples in this book without restriction.

Original sources can be found on GitHub[3]

We appreciate, but do not require attribution. This will usually include the Title of the book and its authors. For Example: *JavaParser: Visited by Smith, Van Bruggen, Tomassetti*

How to Contact Us

Any questions, comments or concerns please come and visit us on Gitter.

Feedback about the book: https://gitter.im/javaparser/javaparserbook

JavaParser library support: https://gitter.im/javaparser/javaparser

[3] https://github.com/javaparser/javaparser-visited

The Authors

Nicholas Smith

Nicholas is a seasoned Software Engineer and Architect currently applying his trade in London. He has been fortunate enough to travel a little with his work as well, working in Portugal, Denmark, India and Italy during his career.

His educational background is principally in focused on Software Engineering, with both a Bachelors and Masters degree, the latter specialising in Financial Services. Perhaps more unusually he has a degree in Wildlife Conservation and a love of red pandas.

Although he has dabbled in several languages in his time he considers himself principally a Java programmer, having used the language since 1.3

Federico Tomassetti

Federico is an independent Software Architect focusing on building languages: parsers, compilers, editors, simulator and other similar tools. He shares his thoughts on language engineering on his blog at tomassetti.me[4].

Previously he got a PhD in Language Engineering, worked here and there (including TripAdvisor and Groupon) while now he focuses on consulting and contributing to open-source projects like JavaParser and JavaSymbolSolver.

Acknowledgements

We would like to thank the original authors of JavaParser: Sreenivasa Viswanadha and Júlio Vilmar Gesser.

Additionally, but most importantly we would like to thank all the contributors and the wonderful community which had grown around JavaParser. Today there are dozens of people who have sent patches, hundreds who have asked questions on forums, Stack Overflow, on GitHub and on Gitter. Discussions with them have helped immeasurable with understanding how JavaParser was used and which parts required clarifications.

[4] http://tomassetti.me

A Brief Introduction to Abstract Syntax Trees

It is not really possible to consider how a Java language parser will be useful without first understanding the concept of syntax trees.

Let's start by trying to imagine that source code can be represented as a tree. Both have a single starting point, from here branches form independently from one another, either as code statements or in case of a tree, actual limbs. As with a real tree's branches, the process of breaking complex statements into smaller parts will continue until we hit a terminal or a leaf.

A tree representing Java code will have a root representing a whole file, with nodes connected to the root for all the top elements of a file, like import or class declarations. From a single class declaration, we could reach multiple nodes, representing the fields or the methods of the class.

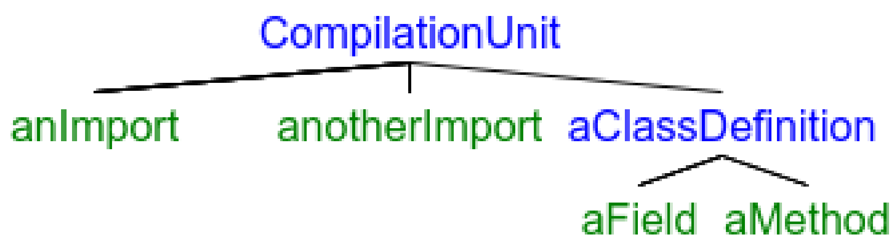

Fig. 1.1 - A first glance to an Abstract Syntax Tree

In the same way that a branch on a real tree has no connection to other branches other than through its origin, the same is true for a tree that represents source code. As a human, or a compiler you understand that a variable reference or a method call relates to another part of the source code, a syntax tree does not, and this is an important distinction[5].

Everything is a node

The tree metaphor only takes us so far, however.

In the classical computer science sense when talking about graphs with vertices and edges, or directed graphs in the case of trees. Depending on the literature you're reading these can also be synonymous with *nodes* and *relationships*. For the purpose of this book, we use **node** as is it considered less convoluted, and in line with the library classes.

[5]Adding the connections between related elements, like method calls to method declarations or field references to a field is the job done by a Symbol Solver. The JavaSymbolSolver is the Symbol Solver created to work with JavaParser.

There are a few simple rules that govern what is a tree. With the exception of the root node, all nodes will have exactly one incoming relationship. The root node will have zero. All nodes can have zero to many outgoing relationships. Those nodes with zero outgoing relationships are considered to be *leaf* nodes and are not considered *parents*. Conversely, those nodes with outgoing relationships are considered to be both a *branch* and a *parent* of one or more *child* nodes.

When a node has children, in the case of the object model created by JavaParser, this is simply represented as a `List<Node>` called `childrenNodes` on the `Node` class.

The vocabulary of trees also leans further on family definitions: child nodes with the same parents are considered *siblings*. *Ancestor* and *descendant* are also used when nodes can be related by immediately navigating up or down the tree respectively. Thus all nodes are a descendant of the root node.

In order to model this within the library the Java language concepts you work with `MethodDeclaration`, `Statement`, `Parameter` etc. will ultimately be descendants (in the object orientated sense) of the `Node` class in the JavaParser library. For the most part, JavaParser has tried to remain true to the names defined within the official grammar specification[6] for the naming of the classes.

When is a node, not a node?

Usually when it is a property of another node.

In addition to having a particular type to describe the language concept they represent e.g. `MethodDeclaration` nodes should have properties to describe their characteristics. Line numbers are a good example of this, for all the nodes in an AST, when parsing the library will document the line number of each node.

If we consider a `MethodDeclaration`, it will have child nodes that represent its **name**, its argument **parameters** (if it has them), its **type** i.e. what it returns and it's **body**. What about other language keywords we can use to define a method e.g. final, static, abstract etc? Well, these are defined as a **modifier** property on the `MethodDeclaration` itself.

Could these be separate node entities in their own right, rather than a property? Yes. Like most things however it was a design choice to represent them in this way, and you will find supporters to argue either case.

Growing our first tree

In order to firm up the intuition for what has been a fairly abstract discussion, we're going to look at a graphical representation of what will be created by parsing simple class with JavaParser.

[6] https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html

```
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {

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

Example 1: TimePrinter

The above class resolves to the following tree:

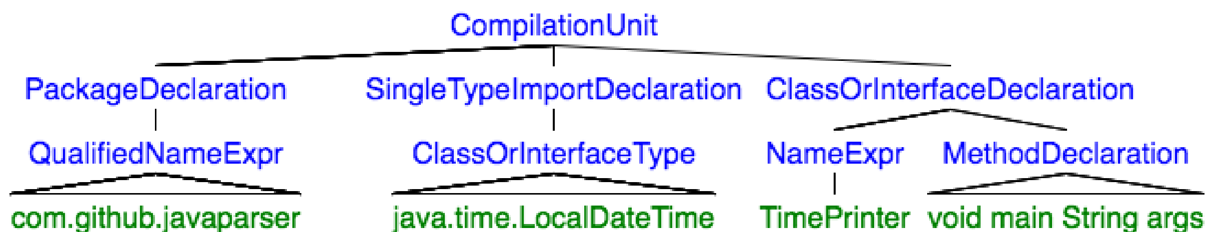

Fig. 1.2 - Compilation Unit

The figure shows that the compilation unit has three children. One for the package declaration, one for an import and the last being for the class declaration. This is not the whole tree, however, consider the triangle notation at the base of the tree above the terminals indicates that this portion has been summarised. Our `ClassOrInterfaceDeclaration` has a child representing the name `NameExpr`, but also a `MethodDeclaration`, it is this in particular that will branch out a lot further.

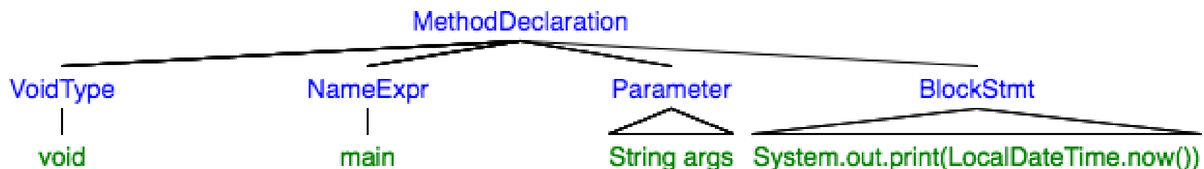

Fig. 1.3 - Method Declaration

We can now see the name, the return type and the parameter(s) for the method along with a `BlockStmt`, which again can be further elaborated.

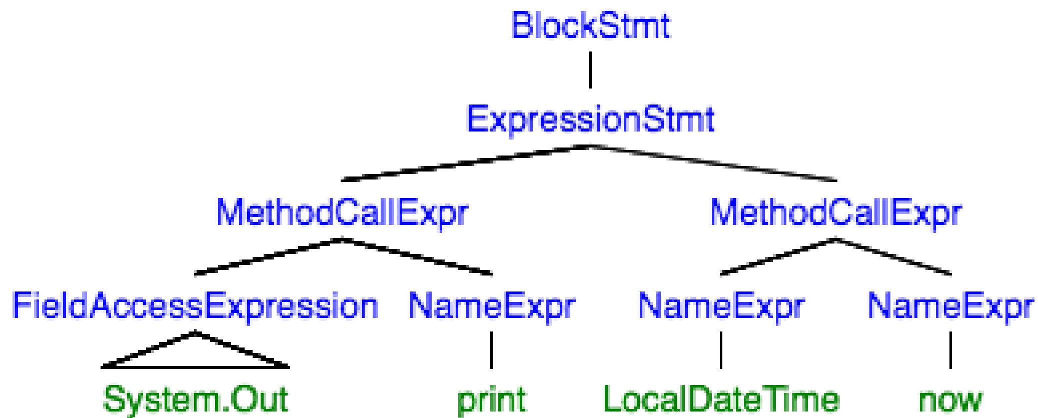

Fig. 1.4 - Block Statement

Finally, but probably most significantly we have the branch of our tree that represents the `BlockStmt`. On its own, it constitutes a significant portion of the tree as a whole, but only represents a single line of code.

We hope you can see from this example that syntax trees can get complex rather quickly!

Beyond Abstract

We should also consider what makes an AST **Abstract** rather than say a Concrete Syntax Tree or Parse Tree. As with its meaning in the wider area of computer science, it is related to the omittance of information. Typically in an AST whitespace and comment tokens are omitted, as are parenthesis which can be inferred by the structure of the tree.

During the support of the JavaParser library, we have however come to the realisation that many people desire the information relating to comments. So by default, while parsing, the JavaParser library will record comment information and include it the tree representation. The ability to exclude comments is possible through configuration options.

Additionally, the team are also working on ways to provide support for whitespace preserving and pretty printing options when writing a class file.

A Flying Visit

In this chapter, we're going to a take a whistle stop tour of the key features the JavaParser library offers. You can think of it as a quick start guide that will get you up on running on your local machine.

We are going to analyze the code of an interesting, although basic calculator that takes input in the Reverse Polish Notation format and calculates the result (wikipedia[7]). It also has a simple memory feature that can store a single number, similar to what you would find a regular calculator.

The class is of a reasonable length so it can be found in Appendix A.

Travelling Companions

Firstly though we should introduce you to your main travelling companions while journeying with the JavaParser.

In addition to the numerous classes provided to represent language concepts, there are a few principal classes that support working with Java AST's namely the `JavaParser`, `StaticJavaParser` and `CompilationUnit`, in addition, there are a variety of `Visitor` classes. We'll look at all of these in more detail over the course of this book, but here is a brief introduction.

In a few words:

- the `JavaParser` class provides a full API for producing an AST from code
- the `StaticJavaParser` class provides a quick and simple API for producing an AST from code.
- the `CompilationUnit` is the root of the AST
- the `Visitors` are classes which are used to find specific parts of the AST

As always the JavaDoc[8] is your friend when using Java libraries, although this book will endeavour to cover as much as possible it will inevitably skip over some areas.

JavaParser Class

As you might imagine this class allows you to parse Java source code into Java objects you can interact with.

[7]https://en.wikipedia.org/wiki/Reverse_Polish_notation
[8]http://www.javadoc.io/doc/com.github.javaparser/javaparser-core

In most cases, you will be working with complete class files, and in this instance the `.parse` method is overloaded to accept a `Path`, a `File`, `InputStream` and a `String` will return a `CompiliationUnit` for you to work with.

It is also possible to work with source fragments as well, although in order to parse a `String` you will need to know the resulting type to avoid a parsing error.

For example:

```
1 Statement statement = StaticJavaParser.parseStatement("int a = 0;");
```

CompilationUnit Class

The `CompilationUnit` is the Java representation of source code from a complete and **syntactically correct** class file you have parsed.

In the context of an AST as mentioned, you can think of the class as the root node.

From here you can access all the nodes of the tree to examine their properties, manipulate the underlying Java representation or use it as an entry point for a `Visitor` you have defined.

Visitor Classes

Although it is feasible to hand roll code to directly traverse the AST in the `CompilationUnit` this 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.

In most cases you will want to determine what you are looking for, then define a visitor that will operate on it. Using a visitor is very easy to find all nodes of a certain kind, like all the method declarations or all the multiplication expressions.

A Simple Visitor

As mentioned it is usually a good idea to define a Visitor that will traverse the AST and operate on the concepts within the code you are interested in. So it makes good sense that we start with a simple Visitor as our first example.

All the code found in this book is available in the accompanying repository[9], you are welcome to check out the working example from there. We do however recommend writing the code yourself as the best way gain intuition into the subject.

In this example, we are going to create a Visitor that examines all the methods in our class and prints their name and line length to the console. So let's get started!

[9] https://github.com/javaparser/javaparser-visited

First, create a new project in your favourite IDE, and add the JavaParser library to your class path. If you prefer to use a dependency management tool you have just to add a few lines to your build script. For Maven these lines are:

```
1 <dependency>
2     <groupId>com.github.javaparser</groupId>
3     <artifactId>javaparser-symbol-solver-core</artifactId>
4     <version>LATEST</version>
5 </dependency>
```

While if you are using Gradle one line, is enough:

```
1 compile group: 'com.github.javaparser', name: 'javaparser-symbol-solver-core', versi\
2 on: 'LATEST'
```

At the time of writing `3.18.0` was the latest version of the library. If you are having trouble with some of the samples here, please revert to this version.

The above dependency includes both core Java parsing and symbol solving capabilities into your project. This book covers both of these libraries to we suggest you include both.

If you only require parsing capabilities you can change the `artifact` tag to be be `javaparser-core`.

Now it is time to write some code. You can take a look at Appendix A and create the `ReversePolishNotation.java` class. The package will be `org.javaparser.examples`.

Next, create second class, the name and package are up to you. We are going to use `org.javaparser.examples.VoidVisitorStarter.java`.

Now we have taken care of all the setup, it is time to create a `CompilationUnit` i.e. the parsed AST for the `ReversePolishNotation` class. Feel free to copy the code below.

```
1 public class VoidVisitorStarter {
2     private static final String FILE_PATH =
3             "src/main/java/org/javaparser/samples/ReversePolishNotation.java";
4     public static void main(String[] args) throws Exception {
5 
6         CompilationUnit cu = StaticJavaParser
7                 .parse(Files.newInputStream(Paths.get(FILE_PATH)));
8     }
9 }
```

VoidVisitorStarter.java

There are a few lines of code here, so let's go through them:

We create a constant `String` which refers to the path of our Java file. As both files exist within the same project we're using its relative path here rather than from the file system root for brevity.

Inside our `main` method we instantiate a `CompilationUnit` type by calling the static `parse` method on the `StaticJavaParser` class; which takes a `InputStream` using the file path previously defined.

Although there is no meaningful output, if you have completed all the previous steps correctly, executing main should complete successfully. Try it now.

Once we have a compilation unit to work with we can define our first visitor. Our visitor is going to extend a class named `VoidVisitorAdaptor`.

Although that sounds like an odd name we can break in down into its constituent parts. The *Void* means we're not expecting the visit to return anything, i.e. this visitor may produce a side effect, but will not operate on the underlying tree. There are other types of visitors that will do this, which we will come to later.

The *Adaptor* refers to the class being part of an adaptor pattern, which provided default implementations of the `VoidVisitor` interface, which accepts around 90 different types as arguments (see Appendix B for the full list). So when defining your own visitor you will want to override the `visit` method for the type you are interested in. In practice, this means you only need to define the methods for nodes you are interested in (e.g. field declarations); the functions inherited from VoidVisitorAdaptor will handle the recursion through the rest of the compilation unit's abstract syntax tree. If you were instead to implement `VoidVisitor` directly you would have to define dozens of methods, one for each node type: not very practical.

In our case, we're going to override the adaptors visit method that accepts a type of `MethodDeclaration` as an argument.

Define a new inner class that looks like the following:

```
1  private static class MethodNamePrinter extends VoidVisitorAdapter<Void> {
2
3      @Override
4      public void visit(MethodDeclaration md, Void arg) {
5          super.visit(md, arg);
6          System.out.println("Method Name Printed: " + md.getName());
7      }
8  }
```

VoidVisitorComplete.java

So we have our new class called `MethodNamePrinter` that extends the `VoidVisitorAdapter`. This class also takes a type parameter, which relates to the second argument we pass into the visit method. We

will come to why this is useful in the second example, however, as we're not going to use the feature here we have made the parameter type a `Void`.

Once the class is defined we define our overriding implementation of `visit`, we're interested in method declarations so the first argument is of the type `MethodDeclaration`. If we were interested in something else, say occurrences of `ImportDeclaration` we would use that as an argument type instead. The second argument is the `VoidVisitorAdaptors` parameterised type.

Although in this particular example it will not affect the behaviour of our visitor we make a call to `super` to ensure child nodes of the current node are also visited. This is typically recommended as additional visits accrue little overhead, however not visiting the whole tree is more likely inclined to result in unwanted behaviour.

For our implementation, we then use the `getName` method from the `MethodDeclaration` to print out the name of the method to the console.

Lastly, we will need to provide the body to our `main` method, where we instantiate a `MethodNamePrinter` and provide it with the `CompilationUnit` to operate on. In addition to a `null` for the second argument, which we will not be using.

```
1 VoidVisitor<Void> methodNameVisitor = new MethodNamePrinter();
2 methodNameVisitor.visit(cu, null);
```

VoidVisitorComplete.java

If you execute your code you should see the following output:

```
1 Method Name Printed: calc
2 Method Name Printed: memoryRecall
3 Method Name Printed: memoryClear
4 Method Name Printed: memoryStore
```

There we go, that is possibly the simplest use of JavaParser. As we are just printing out method names to the console this works well. What if we are doing something more involved though, like wanting to store entries in a database or consider the result of multiple visitors? We would need to do something better.

This is where the mystery second parameter comes in, and we'll take a look at that next.

A Simple Visitor With State

It is often useful during traversal to be able to maintain a record of what we have seen so far. This may be helpful for our current decision-making process, or we may want to collect all occurrences of something deemed interesting.

The second parameter to the `visit` method is there to offer a simple state mechanism that is passed around during the traversal of the tree. This gives us the option of having the visitor perform an action, the visitor's responsibility will be to collect the items i.e. method names. Then what the invoking class does with the method names collected is then up to them.

Let's have a look at this in practice, create another inner visitor class, this time it will be parameterised with a `List<String>`.

```
1 private static class MethodNameCollector extends VoidVisitorAdapter<List<String>> {
2 
3     @Override
4     public void visit(MethodDeclaration md, List<String> collector) {
5         super.visit(md, collector);
6         collector.add(md.getNameAsString());
7     }
8 }
```

VoidVisitorComplete.java

Previously we referred to the argument as being a state, but that quite often conjures a negative image in people's mind. Therefore it might be better to think of this object as being a *collector* in order to make it more palatable. In our visit method, we then add the `String` representing the name of the collector object rather than printing it.

Ok, so we can now add our second visitor to `main` in order to compare the outcome.

```
1 List<String> methodNames = new ArrayList<>();
2 VoidVisitor<List<String>> methodNameCollector = new MethodNameCollector();
3 methodNameCollector.visit(cu, methodNames);
4 methodNames.forEach(n -> System.out.println("Method Name Collected: " + n));
```

VoidVisitorComplete.java

As you can see we additionally instantiate a `List` to be passed into the `visit` method. We then do the work of printing to console in `main` rather than expecting the visitor to do it.

Our new output when running main should be:

```
Method Name Printed: calc
Method Name Printed: memoryRecall
Method Name Printed: memoryClear
Method Name Printed: memoryStore
Method Name Collected: calc
Method Name Collected: memoryRecall
Method Name Collected: memoryClear
Method Name Collected: memoryStore
```

Although considered a good practice to separate concerns in this way in the second example, we should also be willing to be pragmatic, there is nothing wrong with the first way either. Working as a language engineer you might want to quickly write some disposable code that is only ever used once on a code base. In this case, perhaps, the quicker less clean solution is the way to go.

These are two simple examples, but we hope you can see the great potential there is to expand from here. We could easily adapt this code to collect methods that are over 50 lines long. From the String we use to collect names we could expand to use a DTO[10] style object that collects, name, number of lines and if the method has a JavaDoc. We're then on our way to writing our own source code analyser.

A Simple Modifying Visitor

One of the little known minor changes to the Java language in version 7 was the ability to express numeric literals with underscores in the value.

Although not a killer language feature as its application is fairly narrow, it can be particularly useful for readability.

```
public static int ONE_BILLION = 1000000000;
public static int TWO_BILLION = 2_000_000_000;
```

If you work with a code base where you have to define arbitrary large numeric values you can see this convention has its advantages.

We're going to be using our Reverse Polish Notation class as a basis again, if you have skipped ahead to this part of the book you can find it in Appendix A. It just so happens to have a member defined in the pre Java 7 style.

Create a skeleton class with a main method again, we've named ours `ModifyingVisitorStarter`. As before running this class should yield nothing more than a green bar after successfully parsing the source.

[10]DTO stands for Data Transfer Object. It is an object used to store data.

```java
public class ModifyingVisitorStarter {
    private static final String FILE_PATH
            = "src/main/java/org/javaparser/samples/ReversePolishNotation.java";

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

        CompilationUnit cu = StaticJavaParser
                .parse(Files.newInputStream(Paths.get(FILE_PATH)));
    }
}
```

ModifyingVisitorStarter.java

The previous visitor we created in `VoidVisitorComplete.java` used the `VoidVisitorAdaptor` as a basis, as we did not intend changing the AST. This time, however, we wish to change the underlying AST, so we will create a subtype of the `ModifierVisitor` and call it `IntegerLiteralModifier`.

For simplicity, we're only going to concern ourselves with `Integer` numbers in this example.

```java
private static class IntegerLiteralModifier extends ModifierVisitor<Void> {

    @Override
    public FieldDeclaration visit(FieldDeclaration fd, Void arg) {
        super.visit(fd, arg);
        return fd;
    }
}
```

ModifyingVisitorComplete.java

As before the class is parameterised and, as in our first example, we're not concerned with carrying state between the traversal so we will just make it a `Void`.

The difference we see from our previous examples with the `VoidVisitorAdapter` is that our visit method now has a return type `FieldDeclaration`. This matches our first argument type to the `visit` method also, essentially meaning this visitor will replace like for like one field declaration in place of another.

We have decided to operate on `FieldDeclaration`(s) i.e. class members rather than an instance variable. Although you could conceivably define instance variables that are large numbers those are probably an indication of *Magic Numbers* within your code, and you probably need a different form of remediation e.g. variable extraction.

In our initial example above we return the unmodified `FieldDeclaration` object. Once again, running `main` should execute successfully without noticeable results.

Next, in order to achieve our goal of adding underscores to the literal representations we're going to define a helper method along with a regex.

Add the following regex to your class:

```
1  private static final Pattern LOOK_AHEAD_THREE =
2      Pattern.compile("(\\d)(?=(\\d{3})+$)");
```

ModifyingVisitorComplete.java

Then the helper method

```
1  static String formatWithUnderscores(String value){
2      String withoutUnderscores = value.replaceAll("_", "");
3      return LOOK_AHEAD_THREE.matcher(withoutUnderscores).replaceAll("$1_");
4  }
```

ModifyingVisitorComplete.java

This gives us a simple `String` processor that after removing any existing underscores will insert an underscore every third character. For our example with integers this is sufficient; if we intended our visitor to work on floating point numbers then we would also have to consider processing for periods, exponents etc.

Now our helper method is defined lets flesh out our `visit` method, in between the call to `super` and the `return` statement add the following lines:

```
1  fd.getVariables().forEach(v ->
2          v.getInitializer().ifPresent(i ->
3                  i.ifIntegerLiteralExpr(il ->
4                          v.setInitializer(formatWithUnderscores(il.getValue()))
5                  )
6          )
7  );
```

ModifyingVisitorComplete.java

The first thing we do here is iterate over the variables in our field declaration. If this feels odd, remember although seldom seen, we can declare variables in Java with `int a, b, c;`

Next, we're going to ascertain if the variable is initialised with a value. The `getInitializer` method returns a Java `Optional`, so we use the `ifPresent` method, passing in a lambda to execute in the true case. In turn, this checks if the variable is an instance of an `IntegerLiteralExpr`. We could expand this out to cover other numeric values as desired, for a complete case.

Once we know we're dealing with an Integer literal we set the initial value to the result of the current value, having been processed by our underscore formatting method.

Now we have defined our visitor, we can update the `main` method to instantiate it and invoke the visit method with the compilation unit.

```
1 ModifierVisitor<?> numericLiteralVisitor = new IntegerLiteralModifier();
2 numericLiteralVisitor.visit(cu, null);
```

ModifyingVisitorComplete.java

We can then use the `toString` on `CompilationUnit` to print the classes source code to the console.

```
1 System.out.println(cu.toString());
```

ModifyingVisitorComplete.java

Scrolling through the console output you should see our modified literal value.

```
1 // What does this do?
2 public static int ONE_BILLION = 1_000_000_000;
3
4 private double memory = 0;
```

ModifyingVisitorComplete.java

Success! Our code is now improved, it is easier to read when someone comes along after us.

A Simple Comment Reporter

For what is now a pleasant change of pace from the previous examples we're not going to create a Visitor. Although we will start out with our obligatory `main` method, we're then going to use the `getAllContainedComments` method on the `CompilationUnit`. This will provide us with an output of all the comments within our file. We can then go on to use this in some interesting way later.

```java
public class CommentReporterStarter {

    private static final String FILE_PATH =
            "src/main/java/org/javaparser/samples/ReversePolishNotation.java";

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

        CompilationUnit cu = StaticJavaParser
                .parse(Files.newInputStream(Paths.get(FILE_PATH)));

        List<Comment> comments = cu.getAllContainedComments();
    }
}
```

CommentReporterStarter.java

The `Visitor` classes we have worked with so far do not really provide a mechanism to attain all comments, as they operate on specific node types. Comments, on the other hand, tend to cross cut through our code, where their positioning and subsequent *attribution* can be practically anywhere.

Although all `Node` objects have a `getComment` method to be invoked to get a given nodes comments, you would need to traverse the whole tree manually. So with this in mind the library provides as a convenience, a `getAllContainedComments` on the `CompilationUnit` to acquire all comments.

Now that we have a list of comments within our code, let's do something a little more meaningful with them. One of the biggest challenges when working with comments from a parsing point of view is where the comment is attributed to i.e. which node does it document. In some cases, the comment is made on a line where no obvious attribution is clear, and in this case, the parser determines them to be *orphan* comments.

So from our list of comments, we're not only going to report their textual content, their types i.e. if they're a `Block`, `Line` or `JavaDoc` style comment, but also whether or not they're an orphan.

Within our previously created starter class add an inner class that will represent our report items.

```
private static class CommentReportEntry {
    private String type;
    private String text;
    private int lineNumber;
    private boolean isOrphan;

    CommentReportEntry(String type, String text, int lineNumber,
        boolean isOrphan) {
        this.type = type;
        this.text = text;
        this.lineNumber = lineNumber;
        this.isOrphan = isOrphan;
    }

    @Override
    public String toString() {
        return lineNumber + "|" + type + "|" + isOrphan + "|" +
                text.replaceAll("\\n","").trim();
    }
}
```

CommentReporterComplete.java

There is nothing too remarkable here, it is a simple container for the properties of comments we're interested in for our report. The `toString` method creates a pipe delimited representation. It also strips some of the whitespaces out from the comment to make it more console friendly.

Now let's modify our code to create a collection of `CommentReportEntry` items, by mapping the fields we're interested in from the JavaParser `Comment` type.

```
List<CommentReportEntry> comments = cu.getAllContainedComments()
        .stream()
        .map(p -> new CommentReportEntry(p.getClass().getSimpleName(),
                p.getContent(),
                p.getRange().map(r -> r.begin.line).orElse(-1),,
                !p.getCommentedNode().isPresent()))
        .collect(Collectors.toList());

comments.forEach(System.out::println);
```

CommentReporterComplete.java

Here we're using the class' name for the type of our comment, we then use the content for the text. To get the line number we use the `Range` object that all nodes contain. `Range` records position information from the source code, including both the start and end of a line, in addition to column positions. Lastly to ascertain if the comment is an orphan or not we access `getCommentedNode`. `CommentedNode` is a Java `Optional` so a call to `isPresent` to check its existence; if a node does not have a commented node we can consider it to be an Orphan.

In this example, we are operating from the point of view of the comment to identify if it is attributed to a particular node. We can also operate from the node's point of view, from a given node we can use the `node.getComment` method or `node.getOrphanComments`.

Running the application should return an unabridged version of the below:

```
81|BlockComment|true|EOF
7|JavadocComment|false|* A Simple Reverse ...
12|LineComment|false|What does this do?
17|JavadocComment|false|* Takes reverse ...
59|JavadocComment|false|* Memory Recall ...
68|JavadocComment|false|* Memory Clear ...
```

That concludes our brief look at how you can get started using the JavaParser library. Hopefully, your mind is piqued with interesting ways you can delve into your own Java codebases.

Comments - Here Be Dragons

Where does one start on this subject? - Well with a classic programmer metaphor it seems. I would say, as a community of developers, more time has been spent discussing thoughts on comment parsing rather than any other feature within JavaParser project.

It is hard to start perhaps because its tail [bad dragon pun] is not yet over. As of the time of writing the next major release of the library plans on rebuilding the behaviour from the ground up.

Why are comments so challenging to work with? Well, formally comments do not constitute part of the abstract syntax tree for a language. The abstract part meaning that it omits the elements of the syntax tree that are are uninteresting to the compiler.

As such they're not really covered by the Java language specification, beyond a single page; in a document that is 800 pages that barely constitutes as a footnote.

For the most part what you include in comment tokens, and where you include them, becomes "prison rules" i.e. anything goes. Even when you think all eventualities are catered for, someone will come provide a new edge case for where comment parsing falls short.

Unlike annotations that are clearly associated with another language concept, classes, methods, fields, etc. due to their location within the syntax tree, comments are not. In turn, this means the concept of *attribution* i.e. what is commented can become ambiguous; so inevitably the implementation will not suit everyone all of the time.

That all said it is a hugely popular feature of the library.

What does this all mean for you as the user? Well, although we have made the best efforts to make comment parsing and attribution work in a sensible and robust way. It may not meet some peoples expectations with regards to how it works, particularly if you have complex needs.

For example the current behaviour the relationship between a node and comment being 1:1 is not always desirable. Consider:

```
1 // TODO: Fix this
2 // Super important method
3 public void doAllTheThings(){
4     ...
5 }
```

In this case, as a human you can easily understand that the TODO comment is concerned with the method declaration; however, the parser would make this an orphan and only attribute the following line. It is understandable how a user of the library might take a different opinion how this should work.

At this stage in the library's development, we are largely opting to use sensible defaults rather than provide a multitude of configurable options, that in themselves might be more confusing.

In the above case arguably the programmer could have used a multiline comment and we would avoid the issue! In itself that is a good exercise; see if you can identify line comments that appear on consecutive multiple lines and make them into a single block comment.

Comment Attribution

A node can have at most one comment associated with it, which is referenced in its `comment` field. Then the type of comment, as you might expect will be one of either `LineComment`, `BlockComment` or `JavadocComment`.

The relationship between node and comment is bi-directional, so `Comment` maintains a `commentedNode` reference.

When trying to determine where a comment should be attributed, first consider what type of comment it is; block and Javadoc styles behave the same, but line comments are slightly different.

During attribution the comments parser will look to assign block or Javadoc style comments to the following node; either on the same line or subsequent line. Line comments, on the other hand, will be assigned to the node preceding them when they appear on the same line as another node or the following node when they're on a line of their own.

The above only holds true however when the comment is placed 'sensibly' between two tokens.

Orphan Comments

We have touched on this concept in other chapters in the book, but it bears repeating. Comments that can't be attributed to a particular node in the tree are deemed to be orphaned. More often than not, if you can't find a comment attributed to the node you expect, it will be an orphan elsewhere. Most likely in the parent of the node you would expect.

Consider the following:

```
1  // Orphaned 1
2  // Attributed 1
3  public class TimePrinterWithComments {
4
5      // Orphaned 2
6      // Attributed 2
7      public static void main(String args[]){
8          System.out.print(LocalDateTime.now());
9      }
10     // Orphaned 3
11 }
```

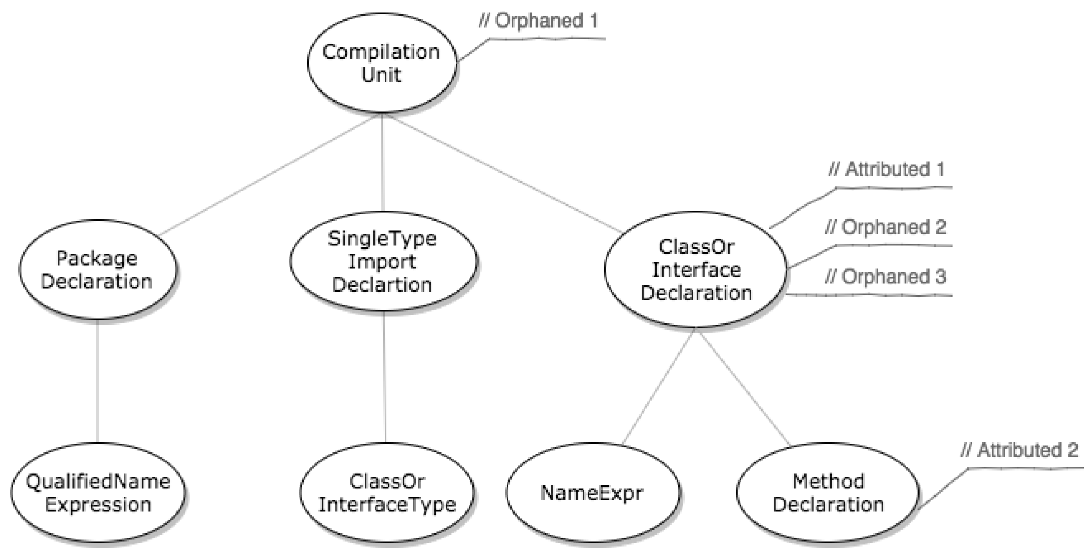

Fig. 3.1 - Comment Attribution

All `Node` objects expose their orphan comments specifically through: `getOrphanComments`; but they will also be returned as part of `getAllContainedComments` .

Working with Comments

As with most cases, we tend to recommend using a visitor to meet your needs, we have previously seen an example of this in our quick start chapter; A Flying Visit. If you have jumped into this chapter you should refer back to that.

Another good option for working with comments in your code aside from visitors is the `getAllContainedComments` method of the `CompilationUnit`. If the current task you're undertaking is primarily concerned with the comments, this gives you a comment centric view of the code. From the list returned you can still navigate `Node` in the AST it is attributed to via the `commentedNode` relationship.

Beyond that, comments themselves are `Node` types, so they're first-class citizens within the compilation unit. They hold largely the same information any AST node would around position. There are some differences; they cannot have comments attributed to them and do not have children, but that should be intuitive. So generally speaking you can work with them in a similar way any other node with respect to examination and manipulation.

Parsing Options

Up until now we have only typically created a `CompilationUnit` by immediately calling the static `parse` method on the `StaticJavaParser` class. This is because the parser is assumed a number of sensible defaults; however, when working with comment parsing there are a couple of dials we can tweak.

Comment parsing is enabled by default, if you decide you wish to disable it, you can provide an instance of a `ParserConfiguration` with `setAttributeComments` to `false`. Doing so will lead to faster parsing, although this will be unnoticeable when working with single source files, but is worth remembering if you are looking to parse large numbers of files.

```
ParserConfiguration parserConfiguration = new ParserConfiguration()
        .setAttributeComments(false);

StaticJavaParser.setConfiguration(parserConfiguration);
```

ConfigurationOptions.java

The second option available to you allows you to control whether or not to consider comments that precede a line space.

Consider the following code:

```
class A {
    // Blah

    boolean f() { return false }
}
```

In this instance the comment `// Blah` will be attributed to the method `f` providing comment attribution is enabled.

Does it really belong to the method, or is it describing what is going to happen in subsequent lines? Like many things, the answer is probably "it depends".

The alternative behaviour offered by JavaParser is to attribute the comment to the containing node. In the above case the for the Class A, and will appear as an entry in the `orphanComments` list.

By default, the parser will use the former behaviour of attributing the comment to the next following node. If you would prefer it to be considered as an orphan, use the following configuration.

```
ParserConfiguration parserConfiguration = new ParserConfiguration()
        .setDoNotAssignCommentsPrecedingEmptyLines(true);
```

ConfigurationOptions.java

Comments In Practice

So here is a fun little class we have put together to exaggerate earlier points around commenting code being "prison rules" where anything goes. Believe it or not, these are actually some of the more sane examples we have seen.

Let's explain the class itself, so it does not distract from what we're actually here for (the comments.)

Mirroring our analogy, we use threads to compete over who will be first/last to write to the console. We define a simple thread class that contains a field that will be each thread's personal catchphrase. When `run` is called on for the thread it will print its catchphrase. The `main` method instantiates three different threads with different catch phrases and then invokes the `start` method. Now the majority of times this code runs it will execute in the manner you would expect, but it is not guaranteed!

Although it's particularly obnoxious to have this variety of comments in a single class, these cases do come up, and can sometimes be counter-intuitive.

```
1  package com.github.javaparser;
2
3  public class PrisonRules /* Maybe use Runnable? */ extends Thread {
4
5      @Deprecated
6      // What is
7      // dibbs?
8      private final String dibbs;
9
10     public PrisonRules(final String dibbs) {
11         this.dibbs = dibbs;
12     }
13
14     @Override // Is this really overridden?
15     public void run() {
16         System.out.println(dibbs);
17     }
18
19     public static void main(String[/* We should pass arguments in here */] args) {
20         Thread t1 = new Thread(new PrisonRules("mine!") /* This should run first */);
21         Thread t2 = new Thread(new PrisonRules("yoink!"));
22         Thread t3 = new Thread(new PrisonRules("finders keepers!"));
23
24         // Here Be Dragons
25         t1/*(^_^)*/.start();
26         t2/*(@_@)*/./*(¬_¬)*/start();
27         /** We need more JavaDocs! */
```

```
28          t3.start();
29      }
30  }
```

PrisonRules.java

For fun we're going to make a quiz out of this, all great learning material needs a quiz. For each comment try and decide which node of the AST the comment will be attributed to. Then we'll go through the answers in more detail afterwards.

1. `/* Maybe use Runnable? */` is attributed to:
    1. The `ClassOrInterfaceDeclaration`
    2. The `ClassOrInterfaceType` "Thread"
    3. The `SimpleName` "PrisonRules"
    4. Nothing, it should be an orphan
2. `// What is` is attributed to:
    1. The `LineComment` "dibbs?"
    2. The `MarkerAnnotationExpr` "Deprecated"
    3. The `FieldDeclaration` "@Deprecated final String dibbs"
    4. Nothing, it should be an orphan
3. `// dibbs?` is attributed to:
    1. The `VariableDeclarator` "dibbs"
    2. The `ClassOrInterfaceType` "String"
    3. The `SimpleName` "dibbs"
    4. Nothing, it should be an orphan
4. `// Is this really overridden?` is attributed to:
    1. The `MarkerAnnotationExpr` "Overriden"
    2. The `MethodDeclaration` "public void run()"
    3. the `VoidType` "void"
    4. Nothing, it should be an orphan
5. `/* We should pass arguments in here */` is attributed to:
    1. The `ArrayType` "String[]"
    2. The `BlockStmt` "{…}"
    3. The `Parameter` "String[] args"
    4. Nothing, it should be an orphan
6. `/* This should run first */` is attributed to:
    1. The `ObjectCreationExpression` "new Thread(…)"
    2. The `OnjectCreationExpression` "new PrisonRules(…)"
    3. It will be an orphan of `ObjectCreationExpression` "new Thread(…)"

   4. It will be an orphan of `ObjectCreationExpression` “new PrisonRules(…)”
7. `// Here Be Dragons` is attributed to:
   1. The `NameExpr` “t1”
   2. The `MethodCallExpr` “t1.start()”
   3. The `ExpressionStmt` “t1.start()”
   4. Nothing, it should be an orphan
8. `/*(^_^)*/` is attributed to:
   1. The `NameExpr` “t1”
   2. The `MethodCallExpr` “t1.start()”
   3. The `SimpleName` “start”
   4. Nothing, it should be an orphan
9. `/*(@_@)*/` is attributed to:
   1. The `NameExpr` “t2”
   2. The `MethodCallExpr` “t2.start()”
   3. The `SimpleName` “start”
   4. Nothing, it should be an orphan
10. `/** We need more JavaDocs! */` is attributed to:
   1. The `MethodDeclaration` “public static void main(…)”
   2. The `MethodCallExpr` “t3.start()”
   3. The `ExpressionStmt` “t3.start()” a4. Nothing, it should be an orphan

Answers:

Question 1, the answer is 1, `ClassOrInterfaceType` “Thread”. When looking to assign a block comment we’re looking for a subsequent node. The `ClassOrInterfaceDeclaration` in this instance is the containing node. The `SimpleName` is the preceding node. The `extends` keyword is modelled through the `extendedTypes` reference on the `ClassOrInterfaceDeclaration` rather than a node itself.

Question 2, the answer is 4, it is an orphan. The `MarkerAnnotationExpr` precedes the line comment on a previous line so it will not be attributed to that. Comment nodes themselves cannot be attributed with comments, so it is not the following `LineComment`. The `FieldDeclaration` starts with the annotation, so is the containing node and where you will find the comment in a `List` referenced by `orphanComments`.

Question 3, the answer is 1, the `VariableDeclarator`, which is the parent of both the `SimpleName` and `ClassOrInterface` types in this instance.

Question 4, the answer is 1, `MarkerAnnotationExpr` “@Overriden”. Unlike `BlockComments`, `LineComments` look for a preceding node on the same line first before considering following nodes.

Question 5, the answer is 4, the comment is defined within the `ArrayType` token; as such it is the parent and where the orphan can be found. There is no node following at this level of the tree to be

attributed to. The `Parameter` node is the parent of the `ArrayType` and although `BlockStmt` follows lexically, is a sibling of `Parameter`.

Question 6, the answer is 3, an orphan of `ObjectCreationExpression` “new Thread..”. For similar reasons to question 5, there are no following nodes at the same level of the tree to attribute the comment to. So it is an orphan comment in the containing node.

Question 7, the answer is 3, `ExpressionStmt` “t1.start()”. When there is no preceding node for a `LineComment` it will look for the next following node. The `NameExpr` and `MethodCallExpr` are both children of the `ExpressionStmt`

Question 8, the answer is 3, `SimpleName` “start”. If you have been paying attention you should have got this one. The `NameExpr` precedes the block comment, so is not considered. The `MethodCallExpr` is the containing parent node. The `SimpleName` “start” is the following node and attributed with the comment.

Question 9, the answer is 4, an orphan. Unlike the previous line of code, there are two comments competing to be attributed to `SimpleName`. The subsequent comment “(¬_¬)” has orphaned it (bad Lenny!), taking `SimpleName` as its parent. Remember - the relationship from nodes to comments is 1:1.

Question 10, the answer is 3, `ExpressionStmt` “t3.start()”. ‘Javadoc comments are attributed in the same way as for block comments; even though there a convention to apply them to Classes, Interfaces, Methods this is not enforced.

Pretty Printing and Lexical Preservation

You can use JavaParser for different purposes. Two of the most common are code generation and code transformation. In both cases you end up generating Java code, probably to store it in a file with the *.java* extension.

A simple call to `toString()` on your `CompilationUnit` will return you a `String` that you can use to write your source. The question we are posed with is: *how will it format the code from your AST?*

The answer is one of two ways:

- using pretty-printing
- using lexical-preserving printing

So let's consider how they differ.

What is Pretty-Printing?

Pretty-printing means that when we write our code out as source, the textual representation is printed in a nicely formatted and standard way.

For example, if you were to parse an existing java class that looks like:

```
   class     A { int



   a;}
```

Then later you decide to pretty-print it using JavaParser you would get following representation of the code:

```
class A {
    int a;
}
```

What is Lexical-Preserving Printing?

Lexical-preserving printing means printing keeping the original layout. When we write code there is information that is not strictly meaningful for the compiler, but is important for human beings. I.e. the position of the whitespace and comments.

If you parse some code and then print it back with Lexical-preserving printing you will get back the same code you parsed. If you parse some code, modify it and then print it back using lexical-preserving printing you will get code very close to the original code, with changes confined to the nodes you have explicitly changed.

You can always use lexical-preserving printing, even on an AST that you have built programmatically and not parsed from source. In this case, there is no original layout to preserve and the code will default to the pretty-printed form. The same is true when you parse code and add to the AST some new nodes created programmatically: the portion of the code that you have parsed will have layout information attached, while the nodes you have created will have not and they will be pretty-printed.

Choosing between Pretty Printing and Lexical-Preserving Printing

We have seen that there are two different styles that can be used when producing code from an AST with JavaParser. The next question to consider is when should we use one, and when should we use the other?

From what we have seen, many cases involve JavaParser being used to perform transformations on large codebases. In this instance, you will typically want to use Lexical-Preserving Printing. The reason being, that all the code that is not modified by your transformation will remain completely unchanged. Typically this is more reassuring to developers, that can get nervous when large (even safe) changes are submitted. This also permits us to have much smaller diff files, making it easy when it comes to manual code reviews. Finally, it allows us to preserve complex layout or formatting that has a meaning.

When generating code or transforming code that is not intended for manual reviews you could instead opt for pretty-printing. Pretty printing is also a good choice if you want to enforce a specific code style: every time you save code you are assured that it is well formed, without any exception.

These are our considerations we consider common but bear in mind that JavaParser and JavaSymbolSolver are intended as tools to be used in many different contexts. You, as the developer, have to analyze your context and make the appropriate choice for you.

We have seen what the differences between pretty-printing and lexical-preserving printing are. Now let's look how to use those and how they work.

Using Pretty Printing

Well, that is actually very simple. You can take every Java node and just call the `toString` method to get the pretty-printed representation of that node.

For example this code:

```
public static void main(String[] args) {
    ClassOrInterfaceDeclaration myClass = new ClassOrInterfaceDeclaration();
    myClass.setComment(new LineComment("A very cool class!"));
    myClass.setName("MyClass");
    myClass.addField("String", "foo");
    System.out.println(myClass);
}
```

PrettyPrintStarter.java

Will produce this output:

```
//A very cool class!
class MyClass {

    String foo;
}
```

It could hardly be any simpler than that.

If you want to have more control you can also use the `PrettyPrinter`. This allows you to configure details like the number of spaces to use for indentation or if you wish to omit comments.

```
public static void main(String[] args) {
    ClassOrInterfaceDeclaration myClass = new ClassOrInterfaceDeclaration();
    myClass.setComment(new LineComment("A very cool class!"));
    myClass.setName("MyClass");
    myClass.addField("String", "foo");

    PrettyPrinterConfiguration conf = new PrettyPrinterConfiguration();
    conf.setIndentSize(1);
    conf.setIndentType(Indentation.IndentType.SPACES);
    conf.setPrintComments(false);
    PrettyPrinter prettyPrinter = new PrettyPrinter(conf);
    System.out.println(prettyPrinter.print(myClass));
}
```

PrettyPrintComplete.java

The output of this program would be this:

```
class MyClass {

 String foo;
}
```

We have:

- removed the comments
- changed the amount of space used for the indentation

These are the most common things that you are likely to want to control when pretty-printing.

You can also customize the `PrettyPrintVisitor` to control which nodes are printed or to do other operations while printing. Let's suppose you want to hide all annotations.

This is how you would achieve that:

```
public static void main(String[] args) {
    ClassOrInterfaceDeclaration myClass = new ClassOrInterfaceDeclaration();
    myClass.setComment(new LineComment("A very cool class!"));
    myClass.setName("MyClass");
    myClass.addField("String", "foo");
    myClass.addAnnotation("MySecretAnnotation");

    PrettyPrinterConfiguration conf = new PrettyPrinterConfiguration();
    conf.setIndentSize(2);
    conf.setIndentType(Indentation.IndentType.SPACES);
    conf.setPrintComments(false);
    Function<PrinterConfiguration, VoidVisitor<Void>> prettyPrinterFactory = (config\
uration) -> new DefaultPrettyPrinterVisitor(conf) {
        @Override
        public void visit(MarkerAnnotationExpr n, Void arg) {
            // ignore
        }

        @Override
        public void visit(SingleMemberAnnotationExpr n, Void arg) {
            // ignore
        }
```

```
23
24          @Override
25          public void visit(NormalAnnotationExpr n, Void arg) {
26              // ignore
27          }
28      };
29      Printer prettyPrinter = new DefaultPrettyPrinter(prettyPrinterFactory, conf);
30      System.out.println(prettyPrinter.print(myClass));
31  }
```

PrettyPrintVisitorComplete.java

Running this code would produce this output:

```
1  class MyClass {
2
3    String foo;
4  }
```

Rather than:

```
1  class MyClass {
2
3    @MySecretAnnotation
4    String foo;
5  }
```

In other words, you have a lot of flexibility with customizing how nodes are pretty-printed; allowing you to implement your pretty-printer in any way that makes sense to you.

Using Lexical-Preserving Printing

Lexical-preserving printing can be used to preserve the original layout the code had when it was parsed. This is its main goal, however you can use it for nodes you have created. In that case, the result will be equivalent to pretty printing those nodes.

To easiest way to set up lexical-preserving printing is to use the `setup` method of the Lexical Preserving Printer. This method will do the parsing, register the initial text and add an observer to the AST. The observer will adapt the text at each modification as you operate on the AST.

In the following example you can see how to parse some code and print it back by using lexical-preserving printing:

```
public static void main(String[] args) {
        String code = "// Hey, this is a comment\n\n\n// Another one\n\nclass A { }";
        CompilationUnit cu = StaticJavaParser.parse(code);
        LexicalPreservingPrinter.setup(cu);
        System.out.println(LexicalPreservingPrinter.print(cu));
}
```

LexicalPreservationStarter.java

At first glance, this code may seem more complex compared to the rest of the JavaParser API. Consider that lexical-preserving printing is a fairly recent addition and we may find ways to simplify the API in the future. Feel free to submit your own suggestions also.

If we were to use the above code to parse and print the following:

```
// Hey, this is a comment


// Another one

class A {}
```

We will get back exactly the same result. That is the whole point of lexical preservation, isn't it?

Well, no, the hard part of implementing lexical preservation is to support the ability to change the AST while preserving and evolving the original layout.

What if we change the name to the class?

```
// use LexialPreservingPrinter.setup
ClassOrInterfaceDeclaration myClass = cu.getClassByName("A").get();
myClass.setName("MyNewClassName");
System.out.println(LexicalPreservingPrinter.print(cu));
```

LexicalPreservationComplete.java

We will get this:

```
// Hey, this is a comment


// Another one

class MyNewClassName {}
```

Then we could add a modifier, to make our class public.

```
ClassOrInterfaceDeclaration myClass = cu.getClassByName("A").get();
myClass.setName("MyNewClassName");
myClass.addModifier(Modifier.Keyword.PUBLIC);
System.out.println(LexicalPreservingPrinter.print(cu));
```

LexicalPreservationComplete.java

The result would be:

```
// Hey, this is a comment


// Another one

public class MyNewClassName {}
```

Finally, what if we wanted to specify the package?

```
ClassOrInterfaceDeclaration myClass = cu.getClassByName("A").get();
myClass.setName("MyNewClassName");
myClass.addModifier(Modifier.Keyword.PUBLIC);
cu.setPackageDeclaration("org.javaparser.samples");
System.out.println(LexicalPreservingPrinter.print(cu));
```

LexicalPreservationComplete.java

This will get you this result:

```
1 package org.javaparser.lexicalpreservation.examples;
2
3 // Hey, this is a comment
4
5
6 // Another one
7
8 public class MyNewClassName {}
```

The idea is that aside from initial setup, you just use the JavaParser normally and behind the scenes, the original text is updated using lexical preservation in a totally transparent way.

You do not need to understand how it works internally unless of course, you are curious; in which case the next section is for you.

How it works: the NodeText and the Concrete Syntax Model

We previously described that the lexical preservation stores the initial text, that is somewhat of a simplification of what happens. What the lexical preservation actually does is create a `NodeText` for each node. A `NodeText` is basically a list of either tokens or placeholders for children.

For example, if you consider this method declaration:

`void foo(int a) { }`

Its `NodeText` would look like this:

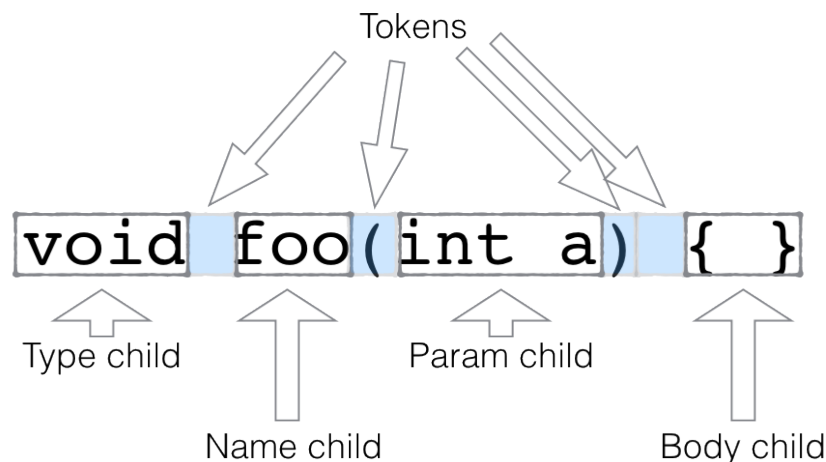

Fig. 6.1 - A NodeText

If we order the different elements:

1. child(void)

2. token(space)
3. child(name)
4. token(lparen)
5. child(param#0)
6. token(rparen)
7. token(space)
8. child(body)

Now suppose that you add a parameter to this method. What will happen? We will need to update the `NodeText` associated to the `MethodDeclaration`.

The Lexical Preservation setup attaches observers to all nodes. At every change, we get notified and we calculate how that node should look like after the change and update these modifications to the `NodeText`.

To understand how the `Node` should look like after the change we use a `ConcreteSyntaxModel`.

What is the Concrete Syntax Model?

You may wonder what the Concrete Syntax Model (CSM) is: it is the definition of what a node should typically look like. The Concrete Syntax Model is what explains to us how to "unparse" an AST Node and transform it into text.

The ConcreteSyntaxModel for a MethodDeclaration looks like this:

```
concreteSyntaxModelByClass.put(MethodDeclaration.class, sequence(
        orphanCommentsBeforeThis(),
        comment(),
        memberAnnotations(),
        modifiers(),
        conditional(ObservableProperty.DEFAULT, FLAG, sequence(token(GeneratedJavaPa\
rserConstants._DEFAULT), space())),
        typeParameters(),
        child(ObservableProperty.TYPE),
        space(),
        child(ObservableProperty.NAME),
        token(GeneratedJavaParserConstants.LPAREN),
        list(ObservableProperty.PARAMETERS, sequence(comma(), space()), none(), none\
()),
        token(GeneratedJavaParserConstants.RPAREN),
        list(ObservableProperty.THROWN_EXCEPTIONS, sequence(comma(), space()),
                sequence(space(), token(GeneratedJavaParserConstants.THROWS), space(\
)), none()),
        conditional(ObservableProperty.BODY, IS_PRESENT,
```

```
20                 sequence(space(), child(ObservableProperty.BODY)), semicolon())
21 ));
```

This is quite complicated because it considers all the potential elements that can compose a `MethodDeclaration` and how to present all of them.

Consider this line, for example:

```
1 list(ObservableProperty.PARAMETERS, sequence(comma(), space()), none(), none()),
```

This tells us that the parameters of the `MethodDeclaration` should be ideally separated by one comma and one space.

Or these lines:

```
1 conditional(ObservableProperty.BODY, IS_PRESENT,
2         sequence(space(), child(ObservableProperty.BODY)), semicolon())
```

This tells us that if the body of the method is present we should print it, with a proceeding space and follow it with a semicolon.

If we use the CSM to calculate how the node should look like before the change we would get this:

1. child(void)
2. child(name)
3. token(lparen)
4. child(param#0)
5. token(rparen)
6. token(space)
7. child(body)

while after the change it would look like this:

1. child(void)
2. child(name)
3. token(lparen)
4. child(param#0)
5. token(comma)
6. token(space)
7. child(param#1)
8. token(rparen)
9. token(space)
10. child(body)

Calculating the difference between CSMs

At this point, a difference between the two computed CSMs is obtained and it basically looks like:

1. keep: child(void)
2. keep: child(name)
3. keep: token(lparen)
4. keep: child(param#0)
5. add: token(comma)
6. add: token(space)
7. add: child(param#1)
8. keep: token(rparen)
9. keep: token(space)
10. keep: child(body)

Finally this difference is applied to the NodeText stored for the Node: we traverse the `NodeText` until we find the point where we have to add or remove new elements. There we apply the changes we need. In this case, we add a comma, a space and a placeholder for a child (the second parameter). Applying the difference is not straightforward because we want to ignore some of the whitespace and comments which are present in the `NodeText`.

Summary

For a long time JavaParser supported only pretty-printing, and for many uses, it was and still is a reasonable choice.

Some users will instead wish to perform modifications while preserving the format of existing code: keeping the exact whitespace, leaving comments exactly where they are. This layout information can be meaningless for a compiler but it is important for developers. For those usages, we created the lexical preservation, introduced for the first time in version 3.1.1.

These two alternatives should cover all of your needs. If not, come talk to us or create a new issue on GitHub.

Solving Symbols and References

JavaParser, like other parsers, take information present in source code and it organizes it into a tree, an Abstract Syntax Tree.

In many situations, the information present in the AST is sufficient for a 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 relations between nodes. In a sense, this means tracing new links and transforming a tree to a graph.

When we find a name used in Java code we recognize it as a *symbol*. Symbol is the term we use to indicate any possible element that could be represented by a name. A symbol could be a variable, a parameter, but also a field or a type.

When we encounter this expression:

```
container.element
```

What is the *container*? What is the *element*? Perhaps *container* is a variable and *element* is a field of that variable. Or maybe *container* is a class name and *element* is a static field of that class. At this stage we do not know, so we speak generally of *symbols*.

The JavaParser Symbol Solver is the part of the library that examines those symbols and tells us what they really are, finding for us the variable declaration, parameter declaration or type declaration that is represented by that symbol.

Consider for example this case:

```
void aMethod() {
   int a;
   a = a + 1;
}
```

By looking at the AST we know that `a = a + 1` corresponds to an assignment. We also know that we are assigning the result of the adding 1 to the value of `a`.

There are things that we do not know:

- Is there a symbol `a` in that context?
- Can that symbol be assigned?
- What is the type of that symbol?

Think about the last point: if `a` is an `int` then the result of `a + 1` is going to be an `int`. However `a` could be a `String`, in that case, we are not summing `1` but we are concatenating its string representation to that the `String` represented by `a`. In other words, by simply looking at the AST we cannot say what kind of operation we are performing: an algebraic sum or a string concatenation? It depends on the type of `a`. To answer these questions we have to resolve references. We have to connect the reference to `a` in `a + 1` to the declaration of `a` that is being referred.

Now consider the following case:

```
1  class Bar {
2  
3      private String a;
4  
5      void aMethod() {
6          while (true) {
7              int a;
8              a = a + 1;
9            }
10     }
11 }
```

Bar.java

There are many `a` symbols. Solving a reference means finding the correct one and it can be difficult. And trust us, things can get very complex. Luckily for you, you can just use the JavaParser Symbol Solver to solve references between elements of a JavaParser AST.

How to setup the JavaParser Symbol Solver?

From this point forward we will refer to the JavaParser Symbol Solver as the 'JavaSymbolSolver', naming things is hard and we don't always get it right first time. Future versions may refactored the class name

JavaSymbolSolver comes bundled with the JavaParser core library. This is because each version of the JavaSymbolSolver is guaranteed to work with a specific version of JavaParser. Historically this was not the case, and the libraries were distributed separately.

If you previously elected not to include the JavaSymbolSolver and just included the `javaparser-core` artifact in your dependencies you will have to update to the bundled version.

```
dependencies {
    compile group: 'com.github.javaparser', name: 'javaparser-symbol-solver-core',
        version: 'LATEST'
}
```

Or if you feel more comfortable with the verbosity of Maven here you go:

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

How to get the type of references: a first example

Let’s consider our initial example. How do we get the type of the value being assigned?

This is the code we want to parse:

```
class Bar {

    private String a;

    void aMethod() {
        while (true) {
            int a = 0;
            a = a + 1;
        }
    }
}
```

Bar.java

This is the code to get the type:

```
public class GetTypeOfReference {

    private static final String FILE_PATH =
            "src/main/java/org/javaparser/examples/chapter5/Bar.java";

    public static void main(String[] args) throws FileNotFoundException {
        TypeSolver typeSolver = new CombinedTypeSolver();

        JavaSymbolSolver symbolSolver = new JavaSymbolSolver(typeSolver);
        StaticJavaParser
                .getParserConfiguration()
                .setSymbolResolver(symbolSolver);

        CompilationUnit cu = StaticJavaParser.parse(new File(FILE_PATH));

        cu.findAll(AssignExpr.class).forEach(ae -> {
            ResolvedType resolvedType = ae.calculateResolvedType();
            System.out.println(ae + " is a: " + resolvedType);
        });
    }
}
```

GetTypeOfReference.java

The output would be: `Type of a = a + 1 is PrimitiveTypeUsage{name='int'}`

This line could look mysterious to you:

```
TypeSolver typeSolver = new CombinedTypeSolver();
```

We'll take a look at what a `TypeSolver` is in the next paragraph.

Specifying where to look for types

In the previous example, we have seen references between elements in the same AST i.e. in the same Java source file. Life tends to be more complex than that and references typically involve other files.

Consider this:

```
class MyClass extends MySuperClass {

    public void returnSomeValue() {
        System.out.println(aField.size());
    }

}
```

Where does `aField` comes from? We do not know, it could be a field defined in `MySuperClass`. Or maybe `MySuperClass` extends another class named `MyExtraSuperClass` and that class defines the field named `aField`.

So we need to check `MySuperClass` to figure it out. Well, this is what JavaSymbolSolver will do for us. In order to do that, however, JavaSymbolSolver will need to know where to look for classes. Classes could be:

- in other Java source files, if they are classes we are compiling right now
- in class files or JARs from some library
- in the JRE, for classes like `java.lang.Object` or `java.lang.String`

Depending on above, there are different `TypeSolvers` that you need to pass to JavaSymbolSolver.

JarTypeSolver: this TypeSolver look for classes inside a JAR file. You just need to specify where the JAR file is located.

JavaParserTypeSolver: this TypeSolver looks for classes defined in source files. You just need to specify the root directory. For example a `src` directory. It will look for source files in a directory corresponding to the package. So it would look for class `a.b.C` in the directory `b` under the directory `a` under the root directory specified.

ReflectionTypeSolver: there are some classes that are defined as part of the language and for which there is no JAR. Classes like `java.lang.Object`. For finding the definition of such classes we can use reflection.

MemoryTypeSolver: This TypeSolver just returns the classes we have recorded in it. It is mostly intended for tests.

CombinedTypeSolver: You want typically to combine multiple TypeSolver into one. To do that you use the CombinedTypeSolver.

Resolving a type using an absolute name

You can use a TypeSolver directly if you want. What this will do is find an element definition given its qualified name.

Consider this example:

```
1   public class UsingTypeSolver {
2
3       private static void showReferenceTypeDeclaration(ResolvedReferenceTypeDeclaratio\
4   n resolvedReferenceTypeDeclaration){
5
6           System.out.printf("== %s ==%n", resolvedReferenceTypeDeclaration
7                   .getQualifiedName());
8           System.out.println(" fields:");
9           resolvedReferenceTypeDeclaration.getAllFields()
10                  .forEach(f -> System.out.printf("    %s %s%n", f.getType(),
11                          f.getName()));
12          System.out.println(" methods:");
13          resolvedReferenceTypeDeclaration.getAllMethods()
14                  .forEach(m -> System.out.printf("    %s%n",
15                          m.getQualifiedSignature()));
16          System.out.println();
17      }
18
19      public static void main(String[] args) {
20          TypeSolver typeSolver = new ReflectionTypeSolver();
21
22          showReferenceTypeDeclaration(typeSolver.solveType("java.lang.Object"));
23          showReferenceTypeDeclaration(typeSolver.solveType("java.lang.String"));
24          showReferenceTypeDeclaration(typeSolver.solveType("java.util.List"));
25      }
26  }
```

UsingTypeSolver.java

In this example we get the definition of three types:

- `java.lang.Object`
- `java.lang.String`
- `java.util.List`

Each of these is an instance of `ReferenceTypeDeclaration`. That class has many methods to access a rich model of the type. You can find all the ancestors, inherited methods and fields, annotations, which types it can be assigned to and more.

Note that it does not matter which TypeSolver you are using: if your class has been found through reflection from parsing a source file or from a JAR, JavaSymbolSolver will provide a model of it with the same interface, i.e. it will always return you an instance of `ReferenceTypeDeclaration`.

Resolving a Type in a context

TypeSolver alone can only find types when their qualified names are provided; however, JavaSymbolSolver can use the context to find which actual qualified name corresponds to a certain declaration. Consider this:

```
1 package org.javaparser.examples.chapter5;
2
3 import org.javaparser.examples.chapter4.*;
4
5 class Foo {
6     Bar bar;
7 }
```

Foo.java

What is the type of the field `bar`?

It could be `org.javaparser.examples.chapter5.Bar` or `org.javaparser.examples.chapter4.Bar`. JavaSymbolSolver will figure this out for you by looking at the context and considering things like:

- Name of the classes in which the type is used
- Import directives
- the current package

How do you tell JavaSymbolSolver which context to use? Well, it determines this by looking at the AST in which the node you are interested into is contained.

```
1 public class ResolveTypeInContext {
2
3     private static final String FILE_PATH
4             = "src/main/java/org/javaparser/examples/chapter5/Foo.java";
5     private static final String SRC_PATH = "src/main/java";
6
7     public static void main(String[] args) throws Exception {
8         TypeSolver reflectionTypeSolver = new ReflectionTypeSolver();
9         TypeSolver javaParserTypeSolver = new JavaParserTypeSolver(
10                new File(SRC_PATH));
11
12        CombinedTypeSolver combinedSolver = new CombinedTypeSolver();
13        combinedSolver.add(reflectionTypeSolver);
```

```
14          combinedSolver.add(javaParserTypeSolver);
15
16          JavaSymbolSolver symbolSolver = new JavaSymbolSolver(combinedSolver);
17          StaticJavaParser
18                  .getParserConfiguration()
19                  .setSymbolResolver(symbolSolver);
20
21          CompilationUnit cu = StaticJavaParser.parse(new File(FILE_PATH));
22
23          FieldDeclaration fieldDeclaration = Navigator.demandNodeOfGivenClass(
24                  cu, FieldDeclaration.class);
25
26          System.out.println("Field type: " + fieldDeclaration.getVariables().get(0)
27                  .getType().resolve().asReferenceType().getQualifiedName());
28      }
29  }
```

ResolveTypeInContext.java

What are we doing here?

- we get the type of the first (and only) variable defined in the FieldDeclaration
- we translate that type to a JavaSymbolSolver type, to do that we pass the JavaParser type and the context to use

The JavaSymbolSolver `ResolvedType` is a type containing all the information you may need on the type.

In this case, the context we use is the `fieldDeclaration`. This means we want JavaSymbolSolver to consider what a reference type `Bar` means when used to point to where that particular node is defined

Consider this:

```
class Foo {
    Bar bar;
}

class Zum {
    Bar bar;

    class Bar {

    }
}
```

From the point of view of JavaParser, both the FieldDeclarations have the same type: a type named `Bar`. But that same name in different contexts means different things. So by passing that same JavaParser type and the context (the first or the second FieldDeclaration) JavaSymbolSolver will give us different answers and figure out the real type being used in each case.

Pretty neat, eh?

Resolving method calls

Another thing that JavaSymbolSolver can help you with is resolving method calls. As you probably know Java permits the overloading of methods, i.e. different methods having the same name but a different signature. This, however, can make it unobvious which method is called by a particular invocation.

Consider this case:

```
class A {

    public void foo(Object param) {
        System.out.println(1);
        System.out.println("hi");
        System.out.println(param);
    }
}
```

A.java

We have three calls and those three calls refer to three different methods that just happen to have the same name.

Here is how we can use JavaSymbolSolver to find out which method is invoked in each case:

```java
public class ResolveMethodCalls {

    private static final String FILE_PATH
            = "src/main/java/org/javaparser/examples/chapter5/A.java";

    public static void main(String[] args) throws Exception {
        TypeSolver typeSolver = new ReflectionTypeSolver();

        JavaSymbolSolver symbolSolver = new JavaSymbolSolver(typeSolver);
        StaticJavaParser
                .getParserConfiguration()
                .setSymbolResolver(symbolSolver);

        CompilationUnit cu = StaticJavaParser.parse(new File(FILE_PATH));

        cu.findAll(MethodCallExpr.class).forEach(mce ->
                System.out.println(mce.resolve().getQualifiedSignature()));
    }
}
```

ResolveMethodCalls.java

This will print:

```
java.io.PrintStream.println(int)
java.io.PrintStream.println(java.lang.String)
java.io.PrintStream.println(java.lang.Object)
```

Note also that JavaSymbolSolver figures out which methods are visible in a certain context, it determines the type of the scope of the method calls (in this case `System.out`) and so on. It is quite some work and you probably do not want to do it yourself, especially when you can just use this library and move on to the next problem.

Using the CombinedTypeSolver

When examining real applications you are typically going to use the `CombinedTypeSolver` to group different other `TypeSolver` into one.

Suppose for example that the application you want to examine uses classes from:

- the standard libraries

- 3 different jars
- 2 directories containing source code

You could create one `CombinedTypeSolver` containing an instance of the `ReflectionTypeSolver`, 3 instances of the `JarTypeSolver` and 2 instances of the `JavaParserTypeSolver`.

The code could look like this:

```
1  public class UsingCombinedTypeSolver {
2  
3      public static void main(String[] args) throws IOException {
4          TypeSolver myTypeSolver = new CombinedTypeSolver(
5                  new ReflectionTypeSolver(),
6                  new JarTypeSolver("jars/library1.jar"),
7                  new JarTypeSolver("jars/library2.jar"),
8                  new JarTypeSolver("jars/library3.jar"),
9                  new JavaParserTypeSolver(new File("src/main/java")),
10                 new JavaParserTypeSolver(new File("generated_code"))
11         );
12         // using myTypeSolver
13     }
14 }
```

UsingCombinedTypeSolver.java

Using the MemoryTypeSolver

As we anticipated this TypeSolver is typically not used in real applications; it is instead mainly intended to be used in tests. Let's consider one real test from the JavaSymbolSolver project:

```
1  public class MemoryTypeSolverComplete {
2  
3      private static final String FILE_PATH =
4              "src/main/java/org/javaparser/examples/chapter5/Foo.java";
5  
6      @Test
7      public void solveTypeInSamePackage() throws Exception {
8          CompilationUnit cu = StaticJavaParser.parse(new File(FILE_PATH));
9  
10         ResolvedReferenceTypeDeclaration otherClass =
11                 EasyMock.createMock(ResolvedReferenceTypeDeclaration.class);
```

```
12         EasyMock.expect(otherClass.getQualifiedName())
13                 .andReturn("org.javaparser.examples.chapter5.Bar");
14
15         /* Start of the relevant part */
16         MemoryTypeSolver memoryTypeSolver = new MemoryTypeSolver();
17         memoryTypeSolver.addDeclaration(
18                 "org.javaparser.examples.chapter5.Bar", otherClass);
19         Context context = new CompilationUnitContext(cu, memoryTypeSolver);
20
21         /* End of the relevant part */
22
23         EasyMock.replay(otherClass);
24
25         SymbolReference<ResolvedTypeDeclaration> ref = context
26                 .solveType("Bar", null);
27         assertTrue(ref.isSolved());
28         assertEquals("org.javaparser.examples.chapter5.Bar",
29                 ref.getCorrespondingDeclaration().getQualifiedName());
30     }
31 }
```

MemoryTypeSolverComplete.java

In this case, we want to verify that the rest of the code looks for one specific type (*com.foo.OtherClassInSamePackage*). Instead of creating a JAR just to use it in the text we use the *MemoryTypeSolver* and instruct it to return a specific *TypeDeclaration* when the name *com.foo.OtherClassInSamePackage* is used.

It could be possible to use it also to analyze applications using special class loaders, generating classes on the fly or in other weird situations. In most cases, you can just ignore the existence of the *MemoryTypeSolver*.

Summary

At this point, you should have an idea of what kind of problems JavaSymbolSolver can solve for you. We do not expect to have answered all your questions on the library and we did not explore the whole API, but we should have provided you with a starting point. The Javadoc can also help you solve some of your remaining questions. For the others feel free to contact us.

Keep in mind JavaSymbolSolver is young compared to JavaParser and its API could evolve to become easier to use over time. However, it is a project which is growing fast, thanks to the wonderful JavaParser community and is already being used in commercial products.

Appendix A - ReversePolishNotation.java

The sample Reverse Polish Notation Calculator source code use in Part 1 examples.

```
1   package com.github.javaparser;
2   
3   import java.util.Stack;
4   import java.util.stream.Stream;
5   
6   
7   /**
8    * A Simple Reverse Polish Notation calculator with memory function.
9    */
10  public class ReversePolishNotation {
11  
12      // What does this do?
13      public static int ONE_BILLION = 1000000000;
14  
15      private double memory = 0;
16  
17      /**
18       * Takes reverse polish notation style string and returns the resulting calculat\
19  ion.
20       *
21       * @param input mathematical expression in the reverse Polish notation format
22       * @return the calculation result
23       */
24      public Double calc(String input) {
25  
26          String[] tokens = input.split(" ");
27          Stack<Double> numbers = new Stack<>();
28  
29          Stream.of(tokens).forEach(t -> {
30              double a;
31              double b;
32              switch(t){
33                  case "+":
```

```
34                      b = numbers.pop();
35                      a = numbers.pop();
36                      numbers.push(a + b);
37                      break;
38                  case "/":
39                      b = numbers.pop();
40                      a = numbers.pop();
41                      numbers.push(a / b);
42                      break;
43                  case "-":
44                      b = numbers.pop();
45                      a = numbers.pop();
46                      numbers.push(a - b);
47                      break;
48                  case "*":
49                      b = numbers.pop();
50                      a = numbers.pop();
51                      numbers.push(a * b);
52                      break;
53                  default:
54                      numbers.push(Double.valueOf(t));
55              }
56          });
57
58          return numbers.pop();
59      }
60
61      /**
62       * Memory Recall uses the number in stored memory, defaulting to 0.
63       *
64       * @return the double
65       */
66      public double memoryRecall(){
67          return memory;
68      }
69
70      /**
71       * Memory Clear sets the memory to 0.
72       */
73      public void memoryClear(){
74          memory = 0;
75      }
76
```

```
77      public void memoryStore(double value){
78          memory = value;
79      }
80
81  }
82  /* EOF */
```

Appendix B - Visitable Nodes

The following provides a quick reference for all the visitable types, for more detailed information please consult the JavaDoc[11] for each class.

JavaParser Type	Source Example
AnnotationDeclaration	**@interface X** { … }
AnnotationMemberDeclaration	@interface X { **int id()**; }
ArrayAccessExpr	getNames()**[15*15]**
ArrayCreationExpr	**new int[5]**
ArrayCreationLevel	new int**[1][2]**
ArrayInitializerExpr	new int[][] **{{1, 1}, {2, 2}}**
ArrayType	**int[][]**
AssertStmt	**assert**
AssignExpr	**a=5**
BinaryExpr	**a && b**
BlockComment	**/* My Comment */**
BlockStmt	**{ … }**
BooleanLiteralExpr	**true**
	false
BreakStmt	**break**
CastExpr	**(long)**15
CatchClause	**catch (Exception e) { … }**
CharLiteralExpr	**‘a’**
ClassExpr	**Object.class**
ClassOrInterfaceDeclaration	**class X { … }**
ClassOrInterfaceType	**Object**
	HashMap<String, String>
	java.util.Punchcard
CompilationUnit	A .java File
ConditionalExpr	**if(a)**
ConstructorDeclaration	X { **X()** { } }
ContinueStmt	**continue**
DoStmt	**do { … } while (a==0)**
DoubleLiteralExpr	**100.1f**
EmptyMemberDeclarationclass	X { ; }
EnclosedExpr	**(1+1)**
EnumConstantDeclaration	X { **A(1)**, B(2) }
EnumDeclaration	**enum X { … }**
ExplicitConstructorInvocationStmt	class X { X() { **super(15)**; } }
	class X { X() { **this(1, 2)**; } }

[11] http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/

JavaParser Type	Source Example
ExpressionStmt	Wraps Expressions into Statements
FieldAccessExpr	**person.name**
FieldDeclaration	**private static int a=15**
ForeachStmt	**for(Object o: objects) { … }**
ForStmt	**for(int a=3,b=5; a<99; a++) { … }**
IfStmt	**if(a==5) hurray() else boo()**
ImportDeclaration	**import com.github.javaparser.JavaParser**
InitializerDeclaration	class X { **static { a=3; }** }
InstanceOfExpr	tool **instanceof** Drill
IntegerLiteralExpr	**8934**
IntersectionType	**Serializable & Cloneable**
JavadocComment	/** a comment */
LabeledStmt	**label123: println("continuing")**
LambdaExpr	**(a, b) -> a+b**
LineComment	**// Comment**
LocalClassDeclarationStmt	class X { void m() { **class Y { }** } }
LongLiteralExpr	**8934l**
MarkerAnnotationExpr	**@Override**
MemberValuePair	@Counters(**a=15**)
MethodCallExpr	**circle.circumference()**
MethodDeclaration	**public int abc() {return 1;}**
MethodReferenceExpr	System.out::**println**
Name	**it.may.contain.dots**
NameExpr	int x = **a** + 3
NodeList	A List of Nodes
NormalAnnotationExpr	**@Mapping(…)**
NullLiteralExpr	**null**
ObjectCreationExpr	**new HashMap.Entry(15)**
PackageDeclaration	**package com.github.javaparser.ast**
Parameter	int abc(**String x**)
PrimitiveType	**int**
ReturnStmt	**return 5 * 5**
SimpleName	**name**
SingleMemberAnnotationExpr	**@Count(15)**
StringLiteralExpr	**"Hello World!"**
SuperExpr	**super**
SwitchEntryStmt	**case 1:**
SwitchStmt	**switch(a) { … }**
SynchronizedStmt	**synchronized (a123) { … }**
ThisExpr	**this**
ThrowStmt	**throw new Exception()**
TryStmt	**try (…) { … } catch (…) { } finally { }**
TypeExpr	**World**::greet
TypeParameter	<**U**> U getU() { … }
UnaryExpr	**11++**

JavaParser Type	**Source Example**
UnionType	catch(**IOException NullPointerException ex**)
UnknownType	DoubleToIntFunction d = **x** -> (int)x + 1
VariableDeclarationExpr	**final int x=3, y=55**
VariableDeclarator	**int x = 14**
VoidType	**void** helloWorld() { ... }
WhileStmt	**while(true) { ... }**
WildcardType	Collection<**?**> c