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 2017-01-19

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.

Contents

Preface

What is JavaParser?

In it's 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 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 library as providing a mechanism the 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 not a symbol solver, it will not answer the question "**where** is this variable defined?" You will need another library for that, for which we recommend the JavaSymbolSolver[1][2].

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.

[1] https://github.com/javaparser/javasymbolsolver

[2] This library is provided by the same team providing JavaParser and it is guaranteed to stay compatible with it.

To labour the point, in the majority of cases when choosing to parsing 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 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[3] 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 trough 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 try our best to maintain the JavaDoc[4] 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 novice, it is recommended that you read the first part in order to gain a good grounding in the subject.

[3] https://docs.oracle.com/javase/specs/jls/se8/html/

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

The second part of the book is a collection of use cases that provide a more detailed look into 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.

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 programatic 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[5]

We appreciate, but do not require attribution. This will usually include the Title of the book and it's 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

[5] http://https://github.com/javaparser/javaparser-visited

The Authors

Nicholas Smith

Nicholas is a seasoned Software Engineer currently applying his trade in London. He has been fortunate enough to travel a little with his work as well, working in 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 later specialising in Financial Services.

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

When he has the free time, other than contributing to the JavaParser project he can be found neglecting is technical blog nicholaspaulsmith.com[6] or hacking some hairbrained scheme that will allow him to retire early.

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[7].

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

Acknowledgements

We would like to thank you 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.

[6] http://nicholaspaulsmith.com
[7] 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 declarations we could reach multiple nodes, representing the fields or the methods of the class.

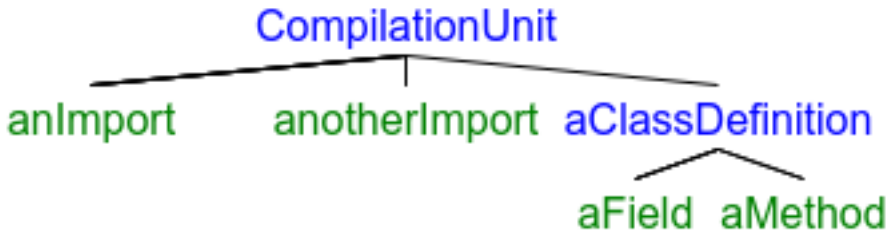

Fig. 1.1 - A first glance to an Abstract Syntax Tree

In the same way that a branch on real tree has has no connection to other branches other than through it's origin, the same it 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[8].

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

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

be synonymous with *nodes* and *relationships*. For the purpose of this book we use **node** as is it considered less convoluted, and inline with the library classes.

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 `childredNodes` 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[9] 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 it's **name**, it's argument **parameters** (if it has them), it's **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.

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

```java
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {

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

The above class resolves to the following tree:

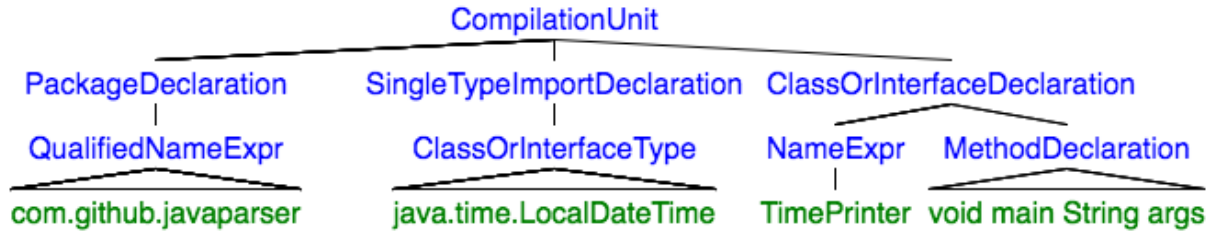

Fig. 1.2 - Compilation Unit

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.

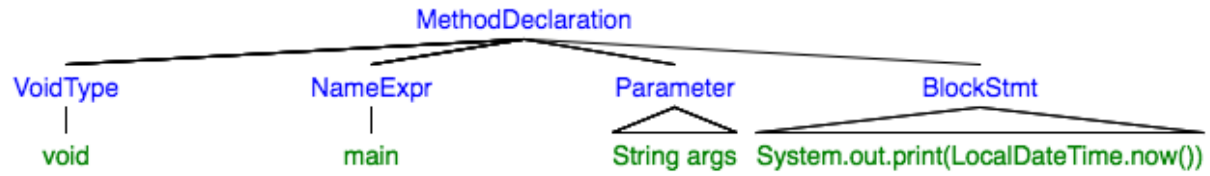

Fig. 1.3 - Method Declaration

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.

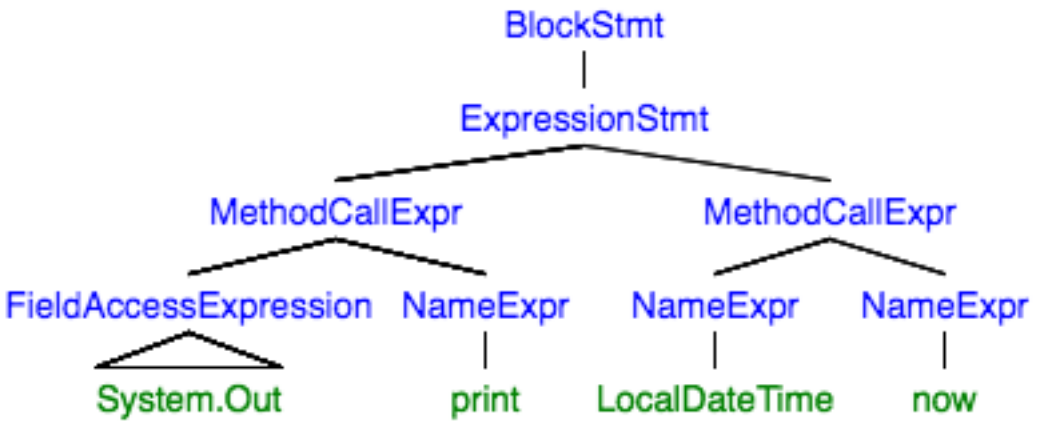

Fig. 1.4 - Block Statement

Finally, but probably most significantly we have the branch of our tree that represents the `BlockStmt`. On it's 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 it's meaning in the wider area of computer science, it 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 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[10]). 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 two principal classes that support working with Java AST's the `JavaParser` 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 brief introduction.

In a few words:

- the `JavaParser` class is what produces the AST from the code
- the `CompilationUnit` is the root of the AST
- the `Visitors` are classes which are useful to find specific parts of the AST

As always the JavaDoc[11] 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.

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.

[10] https://en.wikipedia.org/wiki/Reverse_Polish_notation

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

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 expression = JavaParser.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 a 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[12], you are welcome to checkout 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!

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:

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

```
<dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-core</artifactId>
    <version>3.0.0</version> <!-- Feel free to update the version as needed -->
</dependency>
```

While if you are using Gradle one line is enough:

```
// 3.0.0 is the current version, at the time of writing
compile group: 'com.github.javaparser', name: 'javaparser-core', version: '3.0.0'
```

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. In the example we have gone for `org.javaparser.examples.VoidVisitorExample.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.

```
private static final String FILE_PATH = "src/main/java/org/javaparser/examples/R\
eversePolishNotation.java";

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

        CompilationUnit cu = JavaParser.parse(new FileInputStream(FILE_PATH));

}
```

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 it's 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 `JavaParser` class; which takes a `FileInputStream` using the file path previously defined.

For convenience, although we're creating an instance of a `FileInputStream` we do not need to concern ourselves with closing the resource, the `parse` method will take care of this.

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 it's constituent parts. The *Void* means we're not expecting the visit to return anything, i.e. this visiter may produce a side effect, but 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 that you can defined only the method to handle a certain type of nodes (e.g. field declarations), while your visitor will not do anything with the dozens of other node types you are not interested in. 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 an 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  }
```

So we have our new class called `MethodNamePrinter` that extends the `VoidVisitorAdapter`. This class also takes type parameter, which relates to the second argument we pass into the visit method. We will come onto 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 as we touched on is the `VoidVisitorAdaptors` parameterised type.

Although in this particular example it will not effect 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 the `ComputationUnit` to operate on. In addition to a `null` for the second argument, which we will not be using.

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

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

```
Method Name Printed: calc
Method Name Printed: memoryRecall
Method Name Printed: memoryClear
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 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 rather than as seen our previous example of having the visitor perform an action, the visitors 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.

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

```
private static class MethodNameCollector extends VoidVisitorAdapter<List<String>\
> {

    @Override
    public void visit(MethodDeclaration md, List<String> collector) {
        super.visit(md, collector);
        collector.add(md.getName());
    }
}
```

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 to the collector object rather than printing it.

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

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

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 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 I 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[13] 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 `ModifyingVisitorExample`. As before running this class should yield nothing more than a green bar after successfully parsing the source.

```
public class ModifyingVisitorExample {

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

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

        CompilationUnit cu = JavaParser.parse(in);
    }
}
```

The first two visitors we created previously 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.

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

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

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

As before the class is parameterised and as in out 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 `FielidDeclariton`'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 your 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:

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

Then the helper method

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

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:

```
fd.getVariables().forEach(v ->
        v.getInitializer().ifPresent(i -> {
                if (i instanceof IntegerLiteralExpr) {
                v.setInitializer(formatWithUnderscores(((IntegerLiteralExpr) i).getValue()));
        }
}));
```

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 were 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 more 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 formatter 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.

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

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

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

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

```
// What does this do?
public static int ONE_BILLION = 1_000_000_000;

private double memory = 0;
```

Success! Our the 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 out file. We can then go on to use this in some interesting way later.

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

    CompilationUnit cu = JavaParser.parse(new FileInputStream(FILE_PATH));

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

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 we have a list of comments within out code let's do something a little more meaningful with them. One of the biggest challenges when working with comments from a 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 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 class create 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();
```

```
19      }
20  }
```

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 whitespace out from the comment to make it more console friendly.

Now lets 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().get().begin.line,
                !p.getCommentedNode().isPresent()))
        .collect(Collectors.toList());

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

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 for 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 it's 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 out 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.

Programmatic Refactoring

Content to follow…

Working With Comments

Content to follow…

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 calc\
19 ulation.
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[14] 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)**; } }

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

JavaParser Type	Source Example
	class X { X() { **this(1, 2);** } }
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

JavaParser Type	Source Example
TypeParameter	<**U**> U getU() { ... }
UnaryExpr	**11++**
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