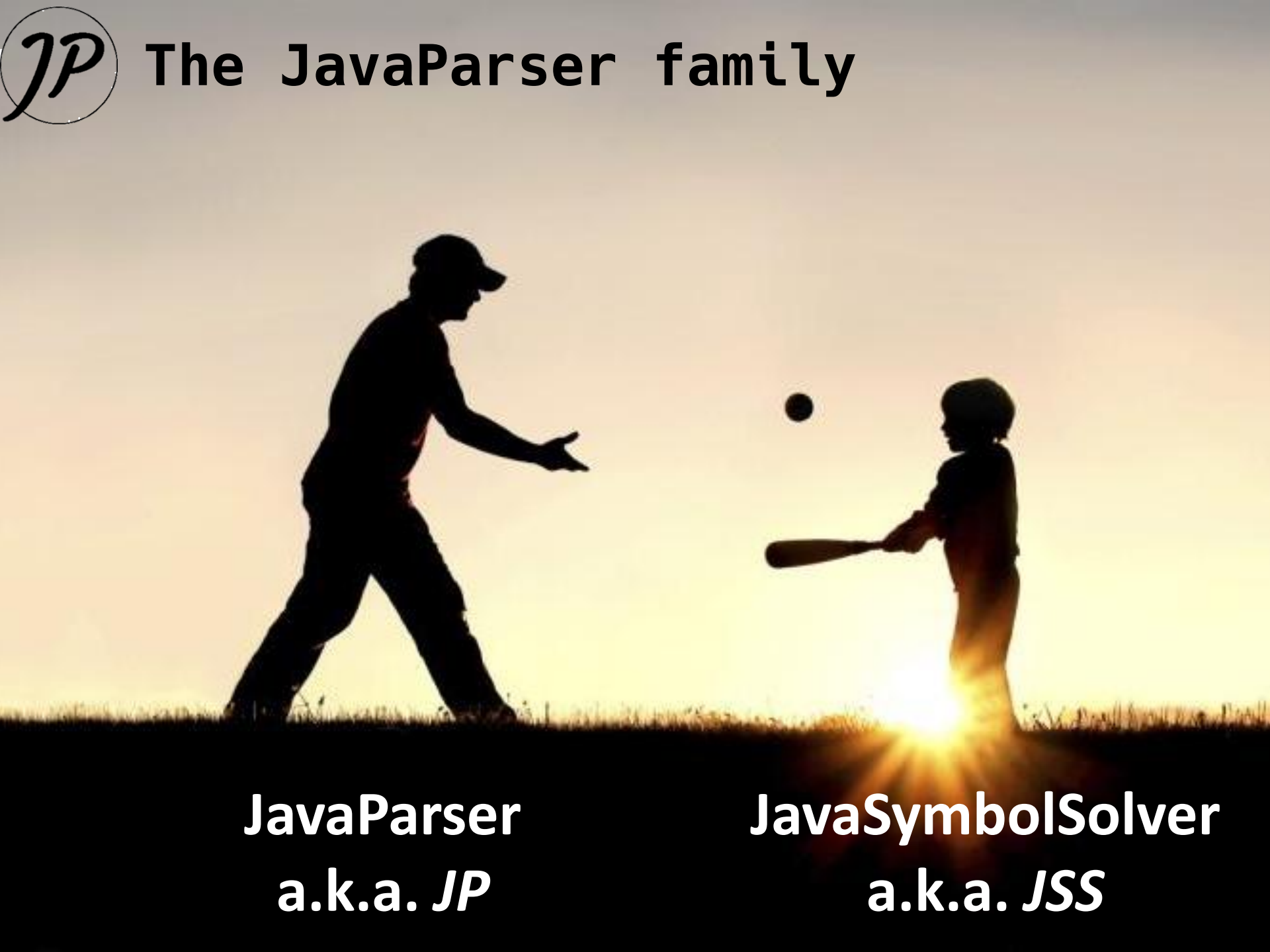
JavaParser

per generare, modificare e analizzare codice Java

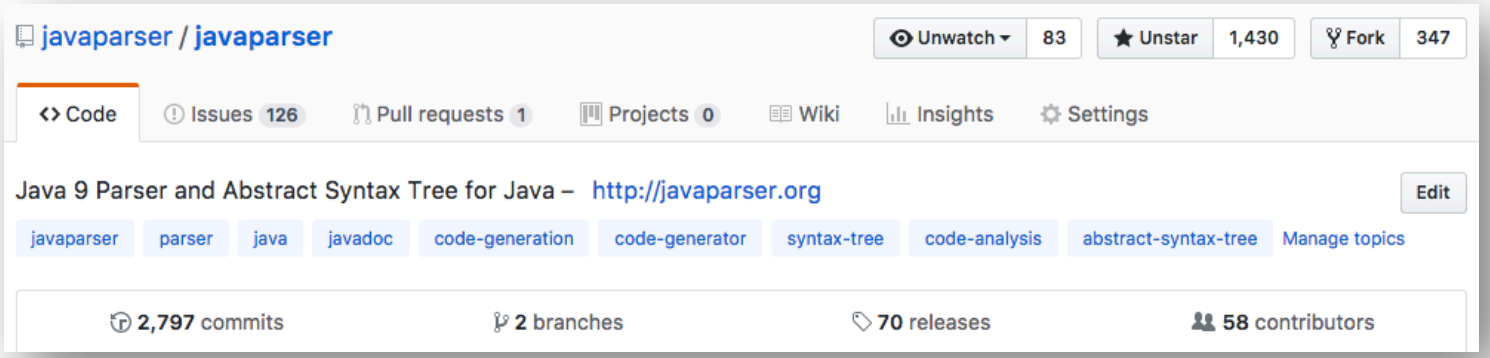
Federico Tomassetti

STRUMENTA

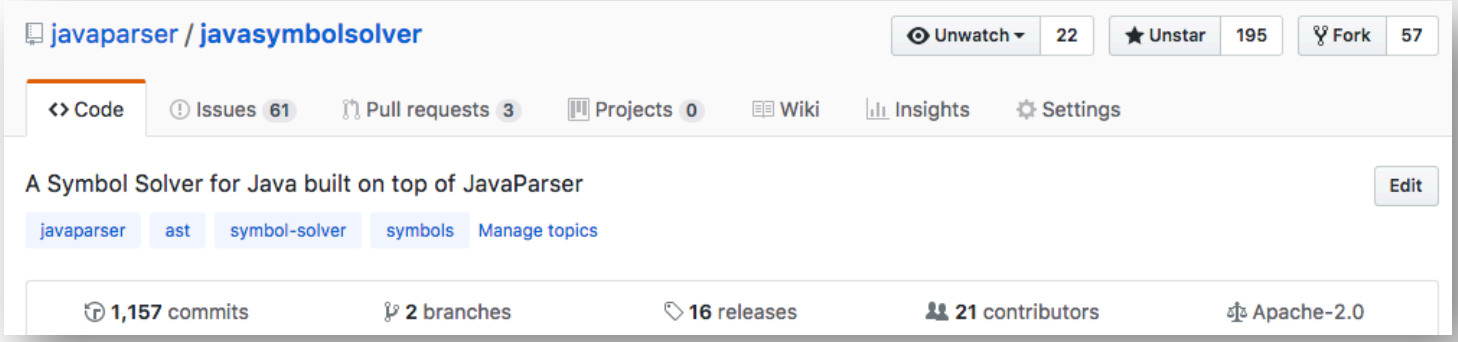

The JavaParser family
JavaParser
a.k.a. JP
JavaSymbolSolver
a.k.a. JSS

Is this stuff mature?

JavaParser is a project with a long history, contributions from over 50 persons, and basically it works.



JavaSymbolSolver is much younger and it works decently enough. Until it does not.



Yes, it supports all of Java

Even the crazy things you all forgot about…

```
public int arrayReturning()[] {
    return new int[]{1, 2, 3, 4, 5};
}
```

Yes, it supports all of Java

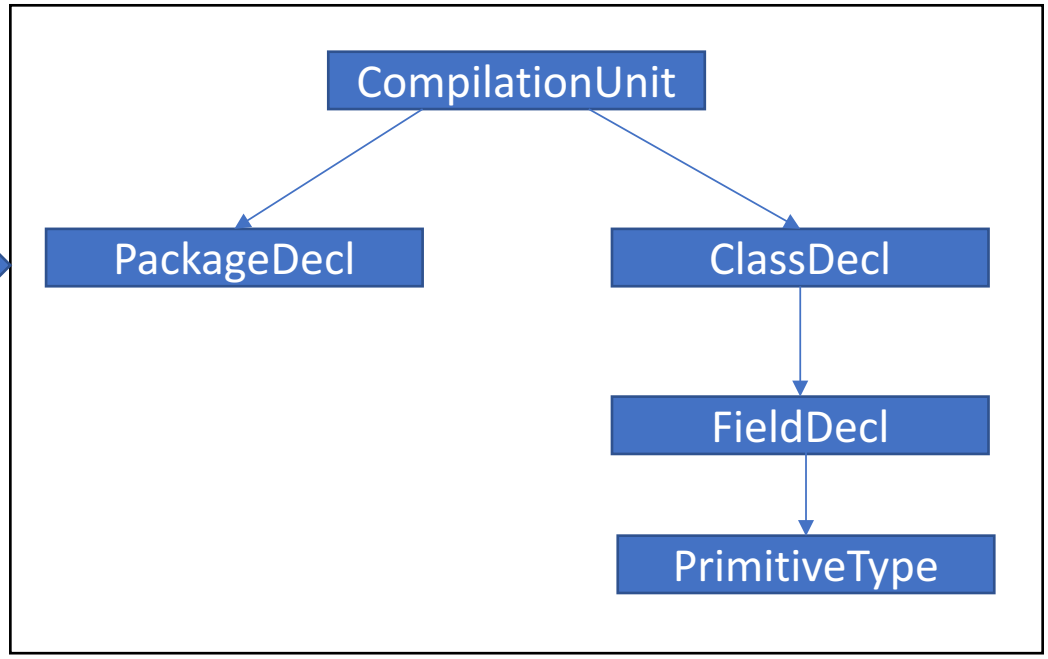
Even the crazy things no one actually used…

```
public void foo(Example this, int a) {
    foo(a);
}
```

What JavaParser does?

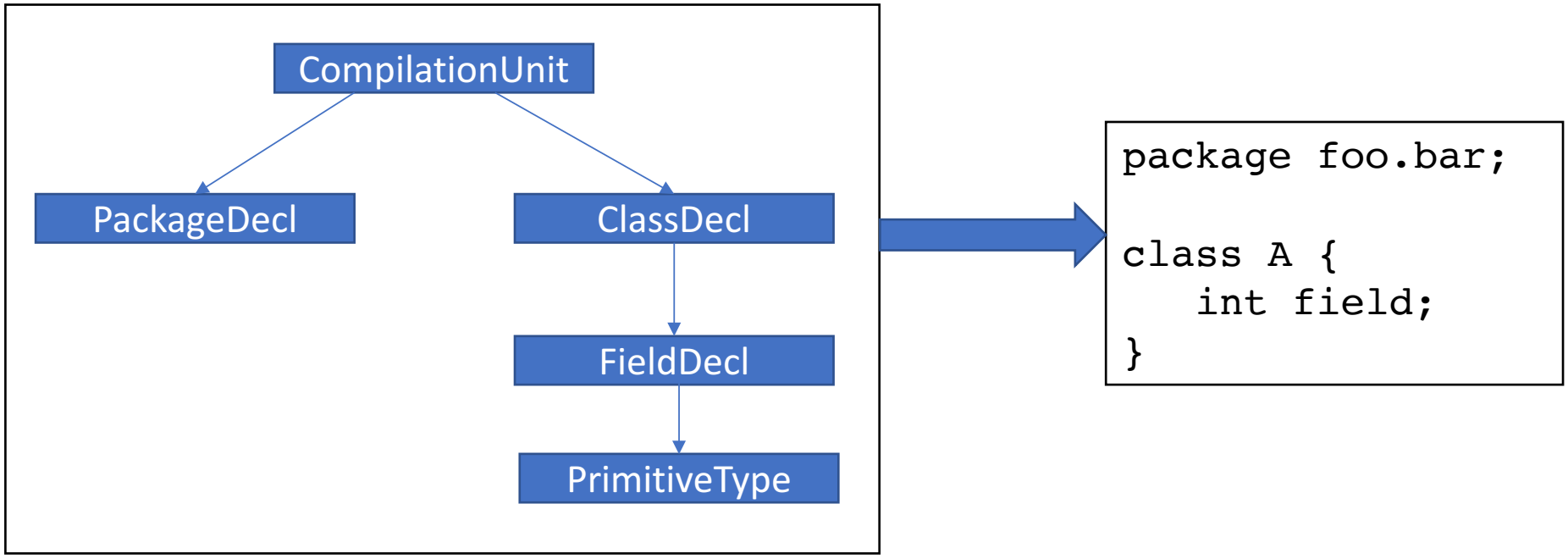
JavaParser… parse Java code into a Java AST

```
package foo.bar;

class A {
   int field;
}
```



What JavaParser does?

JavaParser *unparse* an AST into code



Hello, JavaParser!

```
// Get a compilation unit
JavaParser.parse(myFile)
JavaParser.parse(code)

// Or an expression
JavaParser.parseExpression("1 + 2")

// Or a statement
JavaParser.parseStatement("if (a) b = 1;")
```

Isn't JP enough?

```
int foo;

public void aMethod(int foo) {
    foo = 1;
}

public void anotherMethod() {
    foo = 1;
}
```

To JP these two statements looks the same: they produce the same AST nodes.

It is the assignment of *a thing named "foo", no idea what that thing is*

Isn't JP enough?

```
public void print1(String foo) {
    System.out.print(foo);
}

public void print2(int foo) {
    System.out.print(foo);
}
```

To JP these two statements looks the same: they produce the same AST nodes.

It is the call of *a method named "print", no idea which signature that has*

Isn't JP enough?

```
class A { }

public void creator1() {
   new A();
}

public void creator2() {
   class A { }
   new A();
}
```

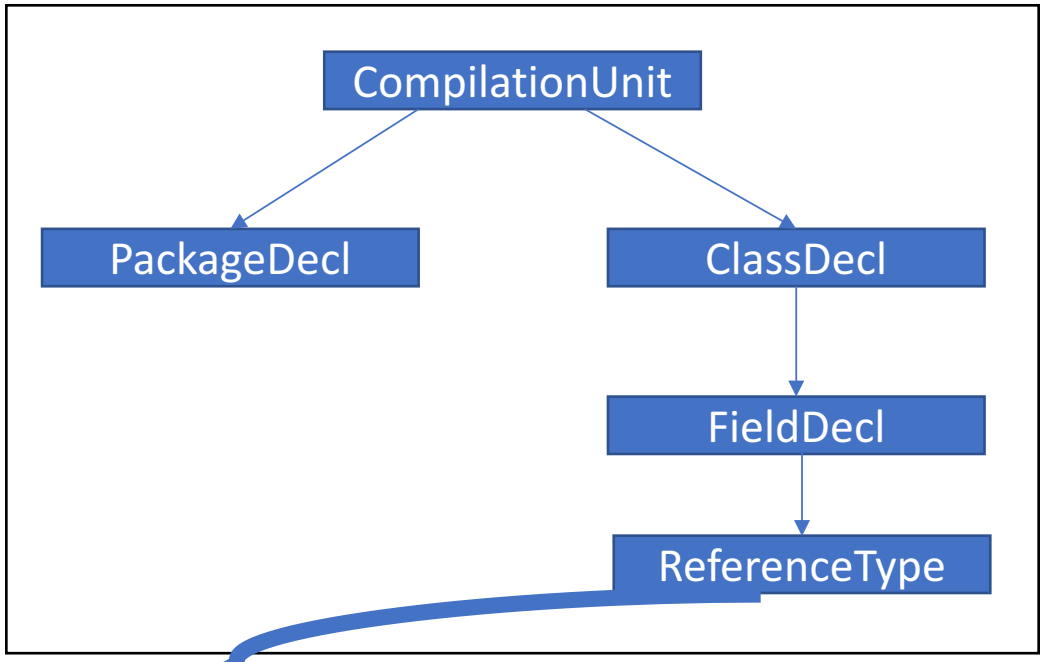
To JP these two statements looks the same: they produce the same AST nodes.

It is the instantiation of *a class named "A", no idea where it is defined*

What JavaSymbolSolver does?

JavaSymbolSolver resolves symbols in the JavaParser AST

```
package foo.bar;

class C {
   D field;
}
```

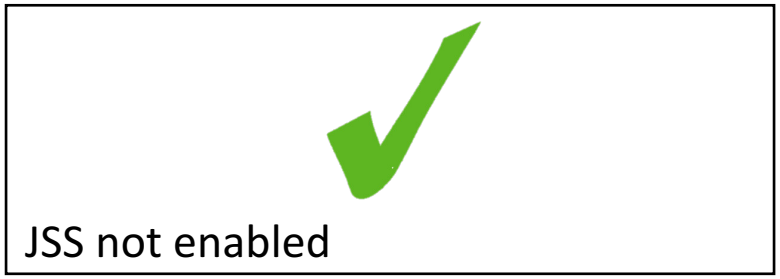


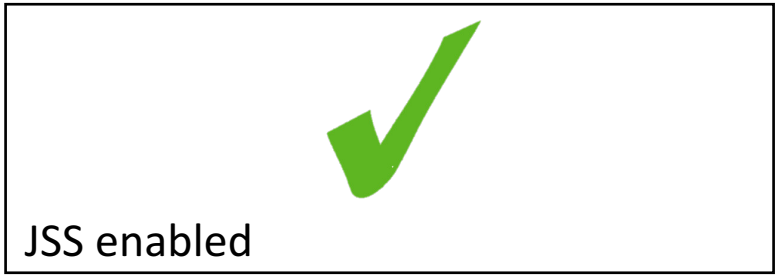

```
package foo.bar;

class D {
}
```

Relationship JP & JSS

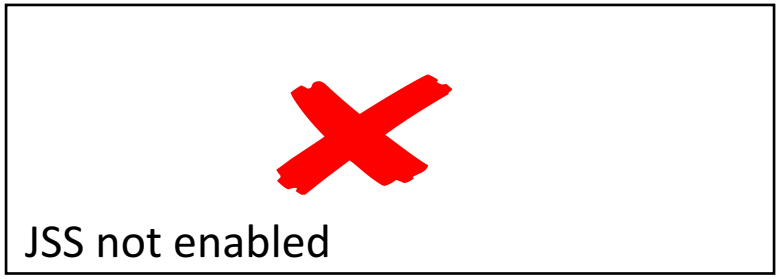
Certain methods in the AST requires additional intelligence.

```
myCallExpression.getName();
```






```
myCallExpression.calculateResolvedType();
```



Hello, JavaSymbolSolver!

```
// 1) Prepare JavaParser
TypeSolver typeSolver = /* configure where to look */;
ParserConfiguration parserConfiguration =
        new ParserConfiguration().setSymbolResolver(
                new JavaSymbolSolver(typeSolver));
JavaParser parser = new JavaParser(parserConfiguration);

// 2) Parse using the advanced API
CompilationUnit compilationUnit =
        parser.parse(ParseStart.COMPILATION_UNIT,
                new StreamProvider(new FileInputStream(myFile)))
          .getResult().get();

// 3) Use the AST… with some extra functionalities
```

Hello, JavaSymbolSolver!

```
CompilationUnit cu = /* we have an AST */


// JSS can calculate the type of any expression
myExpression.calculateResolvedType();
```

myExpression	resolved type
1 + 2	int
2.0 * 3	double
"foo".charAt(0)	char
“foo”.length()	int
new A()	my.packag.A

Hello, JavaSymbolSolver!

```
CompilationUnit cu = /* we have an AST */


// JSS can figure out which method has been called
ResolvedMethodDeclaration methodDeclaration =
                methodCall.resolveInvokedMethod();
methodDeclaration.getQualifiedSignature();
```

methodCall	Method declaration
System.out.print(0)	java.io.PrintStream.print(int)
System.out.print(“a”)	java.io.PrintStream.print(String)
"foo".charAt(0)	java.lang.String.charAt(int)
“foo”.length()	java.lang.String.length()
new LinkedList<String>().size()	java.util.LinkedList.size()
new LinkedList<String>().toString()	java.util.AbstractCollection.toString()

Hello, JavaSymbolSolver!

```
CompilationUnit cu = /* we have an AST */

// JSS knows if two types are assignables
type1 = myExpression.calculateResolvedType();
type2 = fieldDeclaration.resolve().getType();
if (type1.isAssignableBy(type2)) { … }
```

type1	type2	result
int	double	false
double	int	true
Collection<Int>	List<Int>	true
Collection<Double>	List<Int>	false
Collection<? extends String>	List<String>	true

Comments attribution

```
void foo() {
    // comment1
    int a =
        1 + 2; // comment2
}
```

```
    // comment1
    int a =
        1 + 2;
```

```
CompilationUnit cu = JavaParser.parse(code);

ExpressionStmt expressionStmt =
       cu.findFirst(ExpressionStmt.class).get() ;

System.out.println("Comment on the expression statement: "
      + expressionStmt.getComment().get().getContent());
```

Comments attribution

```
void foo() {
    // comment1
    int a =
        1 + 2; // comment2
}
```

```
1 + 2 // comment2
```

```
VariableDeclarationExpr expr =
      (VariableDeclarationExpr)expressionStmt
                                  .getExpression();
VariableDeclarator variableDeclarator =
      expr.getVariables().get(0);
Expression init = variableDeclarator
                    .getInitializer().get();

System.out.println("Comment on the initializer: "
             + init.getComment().get().getContent());
```

Can you show me the AST?

```
Node node = parseBodyDeclaration(
    "public Class<? extends String> methodName(String arg) {}");

// If your grandpa needs the AST
System.out.println(new XmlPrinter(true).output(node));

// Because JavaScript has won
System.out.println(new JsonPrinter(true).output(node));

// Also hipsters need to see an AST
System.out.println(new YamlPrinter(true).output(node));

// To generate a diagram with Graphviz
System.out.println(new DotPrinter(true).output(node));
```

Can you show me the AST?

```
root(Type=MethodDeclaration):
    body(Type=BlockStmt):
    type(Type=ClassOrInterfaceType):
        name(Type=SimpleName):
            identifier: "Class"
        typeArguments:
            - typeArgument(Type=WildcardType):
                extendedType(Type=ClassOrInterfaceType):
                    name(Type=SimpleName):
                        identifier: "String"
    name(Type=SimpleName):
        identifier: "methodName"
    parameters:
        - parameter(Type=Parameter):
            isVarArgs: "false"
            name(Type=SimpleName):
                identifier: "arg"
            type(Type=ClassOrInterfaceType):
                name(Type=SimpleName):
                    identifier: "String"
```

Lexical preservation

JavaParser can do pretty printing

```
String code = "class MyClass{int a;float b;void bar(){}}";
CompilationUnit cu = JavaParser.parse(code);
System.out.println(cu.toString());
```

```
class MyClass {

    int a;

    float b;

    void bar() {
    }

}
```

Lexical preservation

JavaParser can do also do lexical preservation

```
String code = "class MyClass {int a;float b;void bar(){}}";
ParserConfiguration parserConfiguration =
        new ParserConfiguration()
        .setLexicalPreservationEnabled(true);
CompilationUnit cu = new JavaParser(parserConfiguration)
                .parse(ParseStart.COMPILATION_UNIT,
                        new StringProvider(code)
                ).getResult().get();
System.out.println(cu.toString());
```

```
class MyClass {int a; void bar(){}}";
```

Configuring JavaSymbolSolver

JSS needs one thing: that you tell it where to look for classes.

- **CombinedTypeSolver** to group different type solvers
- **AarTypeSolver** look into an aar package
- **JarTypeSolver** look into a jar package
- **JavaParserTypeSolver** look into a directory of Java files
- **MemoryTypeSolver** for testing purposes
- **ReflectionTypeSolver** use reflection (useful to java(x).* classes)

Configuring JavaSymbolSolver

A typical usage:

```
CombinedTypeSolver typeSolver = new CombinedTypeSolver(
        new ReflectionTypeSolver(),
        new JavaParserTypeSolver(new File("src/main/java")),
        new JavaParserTypeSolver(new File("src/test/java")),
        new JarTypeSolver("libs/guava.jar"),
        new JarTypeSolver("libs/log4j.jar"));
```

JavaParser to run queries

Setup: let's consider the code from Hamcrest

```
// The directory where there is the code
File hamcrestCoreDir = new File(
             "src/main/resources/JavaHamcrest-src/hamcrest-
core/src/main/java");

// Configure the Symbol Solver
CombinedTypeSolver typeSolver = new CombinedTypeSolver(
        new ReflectionTypeSolver(),
        new JavaParserTypeSolver(hamcrestCoreDir));

// Use our Symbol Solver while parsing
ParserConfiguration parserConfiguration =
       new ParserConfiguration()
       .setSymbolResolver(new JavaSymbolSolver(typeSolver));
```

JavaParser to run queries

Setup: let's consider the code from Hamcrest

```
// Parse all source files
SourceRoot sourceRoot = new
SourceRoot(hamcrestCoreDir.toPath());
sourceRoot.setParserConfiguration(parserConfiguration);
List<ParseResult<CompilationUnit>> parseResults =
            sourceRoot.tryToParse("");

// Now get all compilation units
List<CompilationUnit> allCus = parseResults.stream()
        .filter(ParseResult::isSuccessful)
        .map(r -> r.getResult().get())
        .collect(Collectors.toList());
```

JavaParser to run queries

Question: *How many methods take more than 3 parameters?*

```
long n = getNodes(allCus, MethodDeclaration.class)
        .stream()
        .filter(m -> m.getParameters().size() > 3)
       .count();
System.out.println("N of methods with 3+ params: " + n);
```

Answer: *11*

JavaParser to run queries

Question: *What are the three top classes with most methods?*

```
getNodes(allCus, ClassOrInterfaceDeclaration.class)
        .stream()
        .filter(c -> !c.isInterface())
        .sorted(Comparator.comparingInt(o ->
             -1 * o.getMethods().size()))
        .limit(3)
        .forEach(c ->
             System.out.println(c.getNameAsString() + ": " +
                   c.getMethods().size() + " methods"));
```

Answer: *CoreMatchers: 35 methods*

BaseDescription: 13 methods

IsEqual: 9 methods

JavaParser to run queries

Question: *What is the class with most ancestors?*

```
ResolvedReferenceTypeDeclaration c = getNodes(allCus,
                                ClassOrInterfaceDeclaration.class)
        .stream()
        .filter(c -> !c.isInterface())
        .map(c -> c.resolve())
        .sorted(Comparator.comparingInt(o ->
                    -1 * o.getAllAncestors().size()))
        .findFirst().get();
List<String> ancestorNames = c.getAllAncestors()
       .stream()
       .map(a -> a.getQualifiedName())
       .collect(Collectors.toList());
System.out.println(c.getQualifiedName() + ": " +
                    String.join(", ", ancestorNames));
```

JSS at work here

Answer: *org.hamcrest.core.StringContains: org.hamcrest.core.SubstringMatcher, org.hamcrest.TypeSafeMatcher, org.hamcrest.BaseMatcher, org.hamcrest.Matcher, org.hamcrest.SelfDescribing, java.lang.Object*

JavaParser to identify patterns

```
private static boolean isClassUsingSingleton(
              ClassOrInterfaceDeclaration c) {
    List<VariableDeclarator> fields = c.getFields()
            .stream()
            .filter(f -> f.isPrivate()
                    && f.isStatic())
            .map(FieldDeclaration::getVariables)
            .flatMap(Collection::stream)
            .filter(v -> isThisClass(c, v.resolve().getType()))
            .collect(Collectors.toList());
    …
}
```

JavaParser to identify patterns

```
private static boolean isClassUsingSingleton(
             ClassOrInterfaceDeclaration c) {
    …
    List<Pair<MethodDeclaration, VariableDeclarator>> pairs =
c.getMethods()
            .stream()
            .filter(m -> m.isPublic()
                    && m.isStatic()
                    && isThisClass(c, m.getType().resolve())
                    && m.getBody().isPresent())
            .filter(m -> fieldReturned(m, fields).isPresent())
            .map(m -> new Pair<>(m, fieldReturned(m,
fields).get()))
            .collect(Collectors.toList());
    return !pairs.isEmpty();
}
```

JavaParser to identify patterns

```
private static boolean isThisClass(
       ClassOrInterfaceDeclaration classOrInterfaceDeclaration,
       ResolvedType type) {
    return type.isReferenceType()
            && type.asReferenceType().getQualifiedName()
                    .equals(classOrInterfaceDeclaration
                            .resolve().getQualifiedName());
}
```

JavaParser to identify patterns

```
private static Optional<VariableDeclarator> fieldReturned(
       MethodDeclaration methodDeclaration,
       List<VariableDeclarator> fields) {
    if (methodDeclaration.getBody().get()
              .getStatements().size() != 1) {
        return Optional.empty();
    }
    Statement statement = methodDeclaration.getBody()
              .get().getStatement(0);
    if (!statement.isReturnStmt() ||
              !statement.asReturnStmt()
                    .getExpression().isPresent()) {
        return Optional.empty();
    }
    …
}
```

JavaParser to identify patterns

```
private static Optional<VariableDeclarator> fieldReturned(
       MethodDeclaration methodDeclaration,
       List<VariableDeclarator> fields) {
    …
    Expression expression = statement.asReturnStmt()
              .getExpression().get();
    if (!expression.isNameExpr()) {
        return Optional.empty();
    }
    Optional<VariableDeclarator> field = fields.stream()
              .filter(f -> f.getNameAsString()
                     .equals(expression.asNameExpr()
                            .getNameAsString()))
              .findFirst();
    return field;
}
```

JavaParser to identify patterns

We are working on the Matcher library to reduce the complexity, it is in the early stages

```
allOf(
        isClass(),
        anyChild(new Binder<>( name: "type",
                        new Binder<>( name: "name",
                                is(FieldDeclaration.class,
                                        f -> f.isPrivate()
                                        && !f.isStatic()
                                        && f.getVariables().size() == 1),
                                f -> ((FieldDeclaration)f).getVariables().get(0)
                                        .getName().getIdentifier()),
                        f -> ((FieldDeclaration)f).getVariables().get(0).getType())),
        anyChild(new Binder<>( name: "type",
                        new Binder<>( name: "name",
                                is(MethodDeclaration.class, m -> m.isPublic() && !m.isStatic()
                                        && m.getParameters().isEmpty()),
                                getterNameToPropertyName),
                        m -> ((MethodDeclaration)m).getType())),
        anyChild(new Binder<>( name: "type",
                        new Binder<>( name: "name",
                                is(MethodDeclaration.class, m -> m.isPublic() && !m.isStatic()
                                        && m.getParameters().size() == 1
                                        && m.getType() instanceof VoidType),
                                setterNameToPropertyName),
                        m -> ((MethodDeclaration)m).getParameter( i: 0).getType()))
));
```

This gives you a list of pairs *name-type* for all the properties in your bean.

JavaParser for automated refactoring

A new version of a library comes up and a deprecated method named *oldMethod* is replaced by *newMethod*. The new method takes 3 parameters: the first one as *oldMethod* but inverted and the third one is a boolean, which we want to be always *true*

```
getNodes(allCus, MethodCallExpr.class)
        .stream()
        .filter(m -> m.resolveInvokedMethod()
                .getQualifiedSignature()
                .equals("foo.MyClass.oldMethod(java.lang.String,
int)"))
        .forEach(m -> m.replace(replaceCallsToOldMethod(m)));
```

JavaParser for automated refactoring

A new version of a library comes up and a deprecated method named *oldMethod* is replaced by *newMethod*. The new method takes 3 parameters: the first one as *oldMethod* but inverted and the third one is a boolean, which we want to be always *true*

```
public MethodCallExpr replaceCallsToOldMethod(
                MethodCallExpr methodCall) {
    MethodCallExpr newMethodCall = new MethodCallExpr(
                methodCall.getScope().get(), "newMethod");
    newMethodCall.addArgument(methodCall.getArgument(1));
    newMethodCall.addArgument(methodCall.getArgument(0));
    newMethodCall.addArgument(new BooleanLiteralExpr(true));
    return newMethodCall;
}
```

JavaParser to generate code

```
CompilationUnit cu = new CompilationUnit();
cu.setPackageDeclaration("jpexample.model");
ClassOrInterfaceDeclaration book = cu.addClass("Book");
book.addField("String", "title");
book.addField("Person", "author");
```

JavaParser to generate code

```
book.addConstructor(Modifier.PUBLIC)
        .addParameter("String", "title")
        .addParameter("Person", "author")
        .setBody(new BlockStmt()
                .addStatement(new ExpressionStmt(new AssignExpr(
                        new FieldAccessExpr(
                                new ThisExpr(), "title"),
                        new NameExpr("title"),
                        AssignExpr.Operator.ASSIGN)))
                .addStatement(new ExpressionStmt(new AssignExpr(
                        new FieldAccessExpr(
                                new ThisExpr(), "author"),
                        new NameExpr("author"),
                        AssignExpr.Operator.ASSIGN))));

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

JavaParser to generate code

```
book.addMethod("getTitle", Modifier.PUBLIC).setBody(
       new BlockStmt().addStatement(
              new ReturnStmt(new NameExpr("title"))));
book.addMethod("getAuthor", Modifier.PUBLIC).setBody(
       new BlockStmt().addStatement(
              new ReturnStmt(new NameExpr("author"))));

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

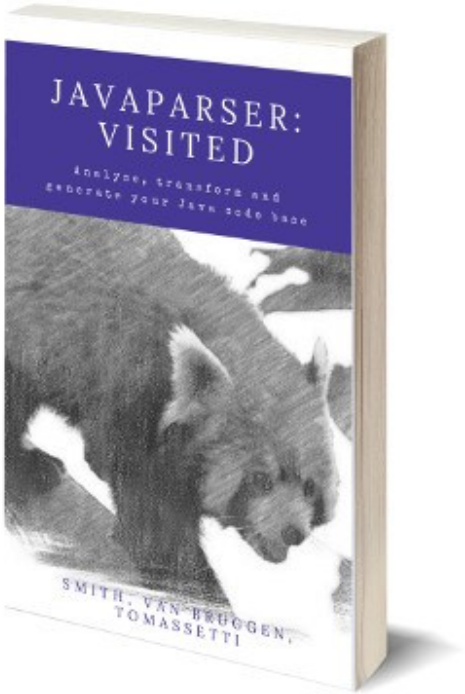
JavaParser to generate code

```
package jpexample.model;

public class Book {
	String title;
	Person author;

	public Book(String title, Person author) {
		this.title = title;
		this.author = author;
	}

	public void getTitle() {
		return title;
	}

	public void getAuthor() {
		return author;
	}
}
```

JavaParser: what can we use it for?

What we can do	Why could we do it
Generate new Java code	Stop writing boilerplate
Modifying existing code	Because large refactoring are boring and error-prone
Running queries on code	So we can answers and data on which to take decision. Also, we can enforce our own rules

JavaParser: Visited



Book on JavaParser and JavaSymbolSolver, from the core committers.

Avalaible for 0+ $

Currently 900+ readers

https://leanpub.com/javaparservisited

Federico Tomassetti

STRUMENTA