Objectives.

1. Support long and double basic types.
2. Support operators.
3. Support conditional expression and switch statement.
4. Support do-while and for statements.
5. Support exception handlers.
6. Support interface type declaration.

In this project, you will only be supporting the parsing of the above programming constructs and their representations in the abstract syntax tree (AST). To compile (just parse for now) your *j--* programs, you need to run the `j--` command as follows:

```
$ $j/j--/bin/j-- -p P.java
```

which will only parse `P.java` and print the AST for the program in XML format.

Note.

1. Consult appendix at the end for the grammar (ie, formal specification) for each new construct you will be supporting in *j--*.
2. The AST output provided for each problem is meant to give you an idea as to what the AST ought to look like once the syntactic constructs for that problem are implemented in *j--*. You are expected to implement the `writeToStdOut()` method in the `J*` files for the constructs such that your AST output is something similar. The autograder will not match your AST against ours for correctness, but instead will test if your parser parses our pass tests without errors.

Problem 1. (*Long and Double Basic Types*) Add support for the `long` and `double` basic types.

```
$ $j/j--/bin/j-- -p tests/BasicTypes.java
```

See `tests/BasicTypes.ast` for output.

Problem 2. (*Operators*) Add support for the following operators, obeying precedence rules (see appendix at the end).

```
~      !=      /       /=       -=
++     --      *=      %        %=
>>     >>=     >>>     >>>=     >=
<<     <<=     <       ^        ^=
|      |=      ||      &        &=
```

```
$ $j/j--/bin/j-- -p tests/Operators.java
```

See `tests/Operators.ast` for output.

Problem 3. (*Conditional Expression*) Add support for conditional expression (`e1 ? e2 : e3`).

```
$ $j/j--bin/j-- -p tests/ConditionalExpression.java
```

See `tests/ConditionalExpression.ast` for output.

Problem 4. (*Switch Statement*) Add support for a switch statement.

```
$ $j/j--/bin/j-- -p tests/SwitchStatement.java
```

See `tests/SwitchStatement.ast` for output.

Problem 5. (*Do-while Statement*) Add support for a do-while statement.

```
$ $j/j--/bin/j-- -p tests/DoWhileStatement.java
```

See `tests/DoWhileStatement.ast` for output.

Problem 6. (*For Statement*) Add support for a for statement.

```
$ $j/j--/bin/j-- -p tests/ForStatement.java
```

See `tests/ForStatement.ast` for output.

Problem 7. (*Exception Handlers*) Add support for exception handling, which involves supporting the `try`, `catch`, `finally`, `throw`, and `throws` clauses.

```
$ $j/j--/bin/j-- -p tests/ExceptionHandlers.java
```

See `tests/ExceptionHandlers.ast` for output.

Problem 8. (*Interface Type Declaration*) Implement support for interface declaration.

```
$ $j/j--/bin/j-- -p tests/Interface.java
```

See `tests/Interface.ast` for output.

Files to Submit

1. `j--.tar.gz` (*j--* source tree as a single gzip file)
2. `report.txt` (project report)

Before you submit:

- Make sure you create the gzip file `j--.tar.gz` such that it only includes the source files and not the binaries, which can be done on the terminal as follows:

```
$ cd $j/j--
$ ant clean
$ cd ..
$ tar -czvf j--.tar j--/*
```

- Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Appendix: Java Syntax

```
compilationUnit ::= [ package qualifiedIdentifier ; ]
                    { import qualifiedIdentifier ; }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= <identifier> { . <identifier> }

typeDeclaration ::= typeDeclarationModifiers ( classDeclaration | interfaceDeclaration )
                  | ;

typeDeclarationModifiers ::= { public | protected | private | static | abstract | final }

classDeclaration ::= class <identifier> [ extends qualifiedIdentifier ]
                       [ implements qualifiedIdentifier { , qualifiedIdentifier } ]
                         classBody

interfaceDeclaration ::= interface <identifier> // can't be final
                           [ extends qualifiedIdentifier { , qualifiedIdentifier } ]
                             interfaceBody

modifiers ::= { public | protected | private | static | abstract | final }

classBody ::= { { ;
                | static block
                | block
                | modifiers memberDecl
                }
              }

interfaceBody ::= { { ;
                    | modifiers interfaceMemberDecl
                    }
                  }

memberDecl ::= <identifier> // constructor
                 formalParameters
                   [ throws qualifiedIdentifier { , qualifiedIdentifier } ] block
             | ( void | type ) <identifier> // method
                 formalParameters
                   [ throws qualifiedIdentifier { , qualifiedIdentifier } ] ( block | ; )
             | type variableDeclarators ; // fields

interfaceMemberDecl ::= ( void | type ) <identifier> // method
                          formalParameters
                            [ throws qualifiedIdentifier { , qualifiedIdentifier } ] ;
                        | type variableDeclarators ; // fields; must have inits

block ::= { { blockStatemnt } }

blockStatement ::= localVariableDeclarationStatement
                 | statement
```

statement ::= block
| `if` parExpression statement [`else` statement]
| `for` `(` [forInit] `;` [expression] `;` [forUpdate] `)` statement
| `while` parExpression statement
| `do` statement `while` parExpression `;`
| `try` block
{ `catch` `(` formalParameter `)` block }
[`finally` block] // must be present if no catches
| `switch` parExpression `{` { switchBlockStatementGroup } `}`
| `return` [expression] `;`
| `throw` expression `;`
| `break` [`<identifier>`] `;`
| `continue` [`<identifier>`] `;`
| `;`
| `<identifier>` `:` statement
| statementExpression `;`

formalParameters ::= `(` [formalParameter { `,` formalParameter }] `)`

formalParameter ::= [`final`] type `<identifier>`

parExpression ::= `(` expression `)`

forInit ::= statementExpression { `,` statementExpression }
| [`final`] type variableDeclarators

forUpdate ::= statementExpression { `,` statementExpression }

switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }

switchLabel ::= `case` expression `:` // must be constant
| `default` `:`

localVariableDeclarationStatement ::= [`final`] type variableDeclarators `;`

variableDeclarators ::= variableDeclarator { `,` variableDeclarator }

variableDeclarator ::= `<identifier>` [`=` variableInitializer]

variableInitializer ::= arrayInitializer | expression

arrayInitializer ::= `{` [variableInitializer { `,` variableInitializer }] `}`

arguments ::= `(` [expression { `,` expression }] `)`

type ::= basicType | referenceType

basicType ::= `boolean` | `byte` | `char` | `short` | `int` | `float` | `long` | `double`

referenceType ::= basicType `[ ]` { `[ ]` }
| qualifiedIdentifier { `[ ]` }

statementExpression ::= expression // but must have side-effect, eg, `i++`

expression ::= assignmentExpression

assignmentExpression ::= conditionalExpression // must be a valid lhs
[
(`=`
| `+=`
| `-=`
| `*=`
| `/=`
| `%=`
| `>>=`
| `>>>=`
| `<<=`
| `&=`
| `|=`
| `^=`
) assignmentExpression]

conditionalExpression ::= conditionalOrExpression [`?` assignmentExpression `:` conditionalExpression]

conditionalOrExpression ::= conditionalAndExpression { `||` conditionalAndExpression }

conditionalAndExpression ::= inclusiveOrExpression { `&&` inclusiveOrExpression }

inclusiveOrExpression ::= exclusiveOrExpression { `|` exclusiveOrExpression }

exclusiveOrExpression ::= andExpression { `^` andExpression }

andExpression ::= equalityExpression { `&` equalityExpression }

equalityExpression ::= relationalExpression { (`==` | `!=`) relationalExpression }

relationalExpression ::= shiftExpression ({ (`<` | `>` | `<=` | `>=`) shiftExpression } | `instanceof` referenceType)

shiftExpression ::= additiveExpression { (`<<` | `>>` | `>>>`) additiveExpression }

additiveExpression ::= multiplicativeExpression { (`+` | `-`) multiplicativeExpression }

multiplicativeExpression ::= unaryExpression { (`*` | `/` | `%`) unaryExpression }

unaryExpression ::= `++` unaryExpression
| `--` unaryExpression
| (`+` | `-`) unaryExpression
| simpleUnaryExpression

simpleUnaryExpression ::= `~` unaryExpression
| `!` unaryExpression
| `(` basicType `)` unaryExpression // basic cast
| `(` referenceType `)` simpleUnaryExpression // reference cast
| postfixExpression

postfixExpression ::= primary { selector } { `++` | `--` }

selector ::= `.` qualifiedIdentifier [arguments]
| `[` expression `]`

primary ::= parExpression
| `this` [arguments]
| `supper` (arguments | `.` `<identifier>` [arguments])
| literal
| `new` creator
| qualifiedIdentifer [arguments]

creator ::= (basicType | qualifiedIdentifier)
(arguments
| `[]` { `[]` } [arrayInitializer]
| newArrayDeclarator
)

newArrayDeclarator ::= `[` [expression] `]` { `[` [expression] `]` }

literal ::= `<int_literal>` | `<char_literal>` | `<string_literal>` | `<float_literal>`
| `<long_literal>` | `<double_literal>` | `true` | `false` | `null`