```
options {
    LOOKAHEAD=1;
}

PARSER_BEGIN(JMM)

import java.util.List;
import java.util.ArrayList;
import pt.up.fe.comp.jmm.report.Report;
import pt.up.fe.comp.jmm.report.ReportType;
import pt.up.fe.comp.jmm.report.Stage;
public class JMM {
    private List<Report> reports = new ArrayList<Report>();

    public static void main(String[] args) throws ParseException {
        System.out.println("Write a Java-- program:");
        JMM jmm = new JMM(System.in);

        SimpleNode root = jmm.Program(); // Returns reference to root node

        System.out.println("Finished parsing");
    }

    public List<Report> getReports() {
        return this.reports;
    }
}

PARSER_END(JMM)

SKIP : {
    " " | "\r" | "\t" | "\n"
}

// Multi-line comment

SKIP : {
    "/*" : WithinComment
}

<WithinComment> SKIP : {
    "*/" : DEFAULT
}

<WithinComment> MORE : {
    <~[]>
}

// Single-line comment

SPECIAL_TOKEN : {
    <SINGLE_LINE_COMMENT: "//" (~["\n", "\r"])* ("\n"|"\r"|"\r\n")>
}

// Keywords

TOKEN : {
    <IMPORT: "import">
    | <CLASS: "class">
    | <EXTENDS: "extends">
    | <PUBLIC: "public">
    | <RETURN: "return">
    | <STATIC: "static">
    | <VOID: "void">
    | <MAIN: "main">
    | <STRING: "String">
    | <TRUE: "true">
    | <FALSE: "false">
    | <INT: "int">
    | <BOOLEAN: "boolean">
    | <IF: "if">
    | <ELSE: "else">
    | <WHILE: "while">
    | <THIS: "this">
    | <NEW: "new">
    | <LENGTH: "length">
}

// Delimiters

TOKEN : {
    <LEFT_PARENTHESES: "(">
    | <RIGHT_PARENTHESES: ")">
    | <LEFT_BRACE: "{">
    | <RIGHT_BRACE: "}">
    | <LEFT_BRACKET: "[">
    | <RIGHT_BRACKET: "]">
    | <SEMICOLON: ";">
    | <COMMA: ",">
}

// Operators

TOKEN : {
    <ADD: "+">
    | <SUB: "-">
    | <MUL: "*">
    | <DIV: "/">
    | <ASSIGN: "=">
    | <DOT: ".">
    | <NOT: "!">
    | <AND: "&&">
    | <LT: "<">
}

// Symbols

TOKEN: {
    <IDENTIFIER:
        (["A"-"Z", "a"-"z", "$"](["0"-"9", "A"-"Z", "a"-"z", "$", "_"])*) |
        (["_"](["0"-"9", "A"-"Z", "a"-"z", "$", "_"])+)
    > |
    <INTEGER: (["0"-"9"])+>
}

SimpleNode Program(): {} {
    ImportDeclaration()
    ClassDeclaration()
    <EOF>
    { return jjtThis; }
}

void ImportDeclaration() #ImportDeclaration: {} {
    (<IMPORT> <IDENTIFIER> (<DOT> <IDENTIFIER>)* <SEMICOLON> )*
}

void ClassDeclaration() #ClassDeclaration : {Token t;} {
    <CLASS> t=<IDENTIFIER> { jjtThis.put("name", t.image); } [ "extends" <IDENTIFIER> ]
    <LEFT_BRACE>
        ClassBody()
    <RIGHT_BRACE>
}

void ClassBody(): {} {
    ( VarDeclaration() )* ( MethodDeclaration() )*
}

void VarDeclaration(): {Token t;} {
    Type() t=<IDENTIFIER> <SEMICOLON>
    { jjtThis.put("name", t.image); }
}

void MethodDeclaration(): {} {
    <PUBLIC>
    (
        (
            <STATIC> <VOID> <MAIN>
            <LEFT_PARENTHESES> <STRING> <LEFT_BRACKET> <RIGHT_BRACKET> <IDENTIFIER> <RIGHT_PARENTHESES> <
LEFT_BRACE>
            MethodBody()
        ) |
        (
            Type() <IDENTIFIER> <LEFT_PARENTHESES>
            [ Type() <IDENTIFIER> ("," Type() <IDENTIFIER>)* ]
            <RIGHT_PARENTHESES> <LEFT_BRACE>
            MethodBody()
            <RETURN> Expression() <SEMICOLON>
        )
    )
    <RIGHT_BRACE>
}

void MethodBody(): {} {
    ( LOOKAHEAD(2) VarDeclaration() )*
    ( Statement() )*
}

void Type() #void : {} {
    ( <INT> [<LEFT_BRACKET> <RIGHT_BRACKET>] )
    | <BOOLEAN>
    | <IDENTIFIER>
}

void Statement() #void : {} {
    ( <LEFT_BRACE> ( Statement() )* <RIGHT_BRACE> )
    | IfStatement()
    | WhileStatement()
    | LOOKAHEAD(2) Assignment()
    | Expression() <SEMICOLON>
}

void IfStatement() #IfStatement : {} {
    <IF> <LEFT_PARENTHESES> Expression() <RIGHT_PARENTHESES>
        Statement()
    <ELSE>
        Statement()
}

void Assignment() #Assignment : {} {
    <IDENTIFIER> [ <LEFT_BRACKET> Expression() <RIGHT_BRACKET> ] <ASSIGN> Expression() <SEMICOLON>
}

void WhileStatement() #WhileStatement : {} {
    <WHILE> (
        <LEFT_PARENTHESES>
        | recover_error(
            new int[]{LEFT_BRACE, INTEGER, TRUE, FALSE, IDENTIFIER, THIS, NEW, NOT, LEFT_PARENTHESES},
            "Got '" + getToken(1).toString() + "' expected '(' token"
          )
    )

    try {
        Expression()
    }

    catch (ParseException e) {
        recover_error(
            new int[] {LEFT_BRACE, INTEGER, TRUE, FALSE, IDENTIFIER, THIS, NEW, NOT, LEFT_PARENTHESES},
            "Found invalid expression '" + e.currentToken + "'"
        );
    }

    (
        <RIGHT_PARENTHESES>
        | recover_error(
            new int[] {LEFT_BRACE, IF, WHILE, INTEGER, TRUE, FALSE, IDENTIFIER, THIS, NEW, NOT,
LEFT_PARENTHESES},
              "Got '" + getToken(1).toString() + "' expected ')' token"
          )
    )

    try {
        Statement()
    }

    catch (ParseException e) {
        recover_error(new int[]{RIGHT_BRACE}, "Found invalid statement '" + e.currentToken + "'");
    }
}

JAVACODE
void recover_error(int[] skipTo, String msg) {
     ParseException e = generateParseException();

     /* System.err.println("Found error on line " + e.currentToken.beginLine + ", column " + e.currentToken.beginColumn);
     System.err.println("\t" + msg);*/

     Report report_error = Report.newError(Stage.SYNTATIC, e.currentToken.beginLine, e.currentToken.beginColumn, msg, e);

     this.reports.add(report_error);

     Token t = getToken(1);

     String skipped_tokens = "";
     boolean match = false;

     while (!match) {
         for (int matcher : skipTo) {
             if (t.kind == matcher) {
                match = true;
                break;
             }
         }

         if (!match) { // skip current token
             skipped_tokens += token.toString() + " ";
             getNextToken();
             t = getToken(1);
         }
     }

     // System.err.println("Tokens skipped " + skipped_tokens);
}

void Expression() #void: {} {
    LT() (<AND> LT() #And(2))*
}

void LT() #void: {} {
    AddSub() (<LT> AddSub() #Less_Than(2))*
}

void AddSub() #void: {} {
    MultDiv() (<ADD> MultDiv() #Add(2) | <SUB> MultDiv() #Sub(2) )*
}

void MultDiv() #void: {} {
    Negate() (<MUL> Negate() #Mul(2) | <DIV> Negate() #Div(2) )*
}

void Negate() #void: {}  {
    (<NOT>)* Length() #Negate
}

void Length() #void: {} {
    Parentheses() (<DOT> Call() )*
```

```
}

void Parentheses() #void: {} {
    ( <LEFT_PARENTHESES> Expression() <RIGHT_PARENTHESES> ) |
    ( <LEFT_BRACKET> Expression() <RIGHT_BRACKET> ) |
    _Expression()
}

void Call(): {} {
    "length" |
    (<IDENTIFIER> <LEFT_PARENTHESES> [ Expression() ( <COMMA> Expression() )* ] <RIGHT_PARENTHESES> )
}

void _Expression() #void: {} {
    (
        <INTEGER> #Int
        | <TRUE> #True
        | <FALSE> #False
        | <IDENTIFIER> #Identifier
        | <THIS> #This
        | NewExpression() #NewExpression
    )
    __Expression()
}

void NewExpression(): {} {
    <NEW>
    (
        <INT> <LEFT_BRACKET> Expression() #IntArray <RIGHT_BRACKET> |
        <IDENTIFIER> #VarCreation <LEFT_PARENTHESES> <RIGHT_PARENTHESES>
    )
}

void __Expression() #Index: {} {
     [ <LEFT_BRACKET> Expression() <RIGHT_BRACKET> __Expression() ]
}

```