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

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


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

// Separators
TOKEN: {
  < COL: "," >
  | < SEMICOL: ";" >
  | < DOT: "." >
  | < LBRACE: "{" >
  | < RBRACE: "}" >
  | < LPAR: "(" >
  | < RPAR: ")" >
  | < LBRACK: "[" >
  | < RBRACK: "]" >
}

// Operations
TOKEN: {
  < AND: "&&" >
  | < OR: "||" >
  | < LESS: "<" >
  | < PLUS: "+" >
  | < MINUS: "-" >
  | < MULT: "*" >
  | < DIV: "/" >
  | < EQ: "=" >
}

TOKEN : {
  < INTEGER_LITERAL: (["0"-"9"])+ >
  | < INTARRAY: "int[]" >
  | < STRINGARRAY: "String[]" >
  | < NOT: "!" >
  | < IDENTIFIER: ["A"-"Z","a"-"z","$","_"](["0"-"9","A"-"Z","a"-"z","_","$"])* >
}

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

void ImportDeclaration(): { Token t; String s = ""; }
{
   <IMPORT> t=<IDENTIFIER> { s += t.image; }
   (<DOT> t=<IDENTIFIER> { s += "."+t.image; } )*
   <SEMICOL> { jjtThis.put("name", t.image); jjtThis.put("qualifiedName", s); }
}

void ClassDeclaration(): { Token t; }
{
   <CLASS> t=<IDENTIFIER> { jjtThis.put("name", t.image); } #Class
   [
      <EXTENDS> t=<IDENTIFIER> {jjtThis.put("extendedClass", t.image); } #Extends
   ]
   <LBRACE>
   (VarDeclaration())*
   (<PUBLIC> (MainDeclaration() | MethodDeclaration()))*
   <RBRACE>
}

void VarDeclaration(): { Token t; }
{
   Parameter() <SEMICOL>
}

void MethodDeclaration(): { Token t; }
{
   Parameter()
   <LPAR>
   [
      Parameter() (<COL> Parameter())*
   ]
   <RPAR>

   <LBRACE>
   (LOOKAHEAD(2) VarDeclaration())*
   (LOOKAHEAD(2) Statement())*
   <RETURN> Expression() #ReturnExpression <SEMICOL>
   <RBRACE>
}

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

void MainDeclaration(): { Token t; }
{
   <STATIC> <VOID> <MAIN>
   <LPAR>
   <STRINGARRAY>
   t=<IDENTIFIER> { jjtThis.put("name", t.image); } #Args
   <RPAR>

   <LBRACE>
   (LOOKAHEAD(2) VarDeclaration())*
   (LOOKAHEAD(2) Statement())*
   <RBRACE>
}


void Type(): { Token t; }
{
   t=<INTARRAY> { jjtThis.put("name", t.image); }
   | t=<BOOLEAN> { jjtThis.put("name", t.image); }
   | t=<INT> { jjtThis.put("name", t.image); }
   | t=<IDENTIFIER> { jjtThis.put("name", t.image); }
}

void Statement() #void: { Token t; }
{
   ifStatement()
   elseStatement()
   |
   whileStatement()
   |
   LOOKAHEAD(2) t=<IDENTIFIER> { jjtThis.put("name", t.image); } #Var <EQ> Expression() <SEMICOL>
#Assignment(2)
   |
   LOOKAHEAD(2) (t=<IDENTIFIER> { jjtThis.put("name", t.image); } #Var exprArray()) <EQ> Expression()
<SEMICOL> #Assignment(2)
   |
   Expression() <SEMICOL>
   |
   <LBRACE>
   (Statement())*
   <RBRACE>
}

void ifStatement(): {}
{
   <IF> <LPAR> Expression() <RPAR> Statement()
}

void elseStatement(): {}
```

```
{
  <ELSE> Statement()
}

void whileStatement(): {}
{
  <WHILE> <LPAR> Expression() <RPAR>
  Statement()
}

void Expression() #void: {}
{
  And()
}

void And() #void: {}
{
  Less() (LOOKAHEAD(2) <AND> Less() #And(2))*
}

void Less() #void: {}
{
  PlusMinus() (LOOKAHEAD(2) <LESS> PlusMinus() #Less(2))*
}

void PlusMinus() #void: { Token t; }
{
  MultDiv()
  (LOOKAHEAD(2) (t=<PLUS> { jjtThis.put("name", t.image); } MultDiv() |
          t=<MINUS> { jjtThis.put("name", t.image); } MultDiv()) #Operation(2))*
}

void MultDiv() #void: { Token t; }
{
  Rest()
  (LOOKAHEAD(2) (t=<MULT> { jjtThis.put("name", t.image); } Rest()|
          t=<DIV> { jjtThis.put("name", t.image); } Rest()) #Operation(2))*
}

void Rest() #void: { Token t;}
{
  (
    IntegerLiteral()
    |
    RestTrue()
    |
    RestFalse()
    |
    RestIdentifier()
    |
    RestThis()
    |
    newAssignment()
    |
    Not()
    |
    ExpBetweenPars()
    ) Terminal()
}

void IntegerLiteral() #void: { Token t; }
{
    t = <INTEGER_LITERAL> { jjtThis.put("name", t.image); } #IntegerLiteral
}

void RestTrue() #void: { Token t; }
{
    t = <TRUE> { jjtThis.put("name", t.image); } #True
}

void RestFalse() #void: { Token t; }
{
    t = <FALSE> { jjtThis.put("name", t.image); } #False
}

void RestThis() #void: { Token t; }
{
    t = <THIS> { jjtThis.put("name", t.image); } #This
}

void Not() #void: { Token t; }
{
    t = <NOT> Expression() { jjtThis.put("name", t.image); } #Not(1)
}

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

void newAssignment() #void: { Token t; }
{
    <NEW>
    (
        ((<INT> <LBRACK> Expression() <RBRACK> #NewArray(1)) )
        |
        (t=<IDENTIFIER> { jjtThis.put("name", t.image); } #NewInstance <LPAR> <RPAR> )
    )
}

void ExpBetweenPars() #void: { Token t; }
{
    <LPAR> Expression() #Exp <RPAR>
}

void exprArray() #ArrayAccess(2): { Token t; }
{
    <LBRACK> Expression() <RBRACK>
}

void Terminal() #void: { Token t; }
{
    exprArray() Terminal()
    |
    dotTerminal()
    |
    {}
}

void dotTerminal() #void: { Token t; }
{
    <DOT>
    (
        (t=<LENGTH> { jjtThis.put("name", t.image); } #DotLength(1) Terminal())
        |
        ((t=<IDENTIFIER> <LPAR>
            InsideFunction()
            <RPAR> { jjtThis.put("name", t.image); } #MethodCall(2))
            Terminal())
    )
}


void InsideFunction(): {}
{
    [
        Expression()
        (<COL> Expression())*
    ]
}
```