options  
{  
 LOOKAHEAD = 1;  
 MULTI = true;  
 NODE\_SCOPE\_HOOK=true;  
}  
  
PARSER\_BEGIN(Parser)  
  
import pt.up.fe.comp.jmm.report.Report;  
import pt.up.fe.comp.jmm.report.ReportType;  
import pt.up.fe.comp.jmm.report.Stage;  
import java.util.ArrayList;  
  
public class Parser {  
  
 int errors = 0;  
 ArrayList<Report> syntacticErrors = new ArrayList<Report>();  
  
 void jjtreeOpenNodeScope(Node n) {  
 ((SimpleNode)n).put("line", Integer.toString(getToken(1).beginLine));  
 ((SimpleNode)n).put("col", Integer.toString(getToken(1).beginColumn));  
 }  
  
 void jjtreeCloseNodeScope(Node n) {  
 }  
  
 ArrayList<Report> getSyntacticErrors() {  
 return syntacticErrors;  
 };  
  
 void errorFunction(ParseException e) {  
 Report error = new Report(ReportType.ERROR,Stage.SYNTATIC,e.currentToken.beginLine, e.currentToken.beginColumn,"Encountered \"" + e.currentToken.image+ "\" " + ".");  
 this.syntacticErrors.add(error);  
  
 this.errors ++;  
  
 Token t = getToken(0);   
 while (true) {  
 if (t.kind == RPAREN && getToken(1).kind != RPAREN) {  
 break;  
 }  
 else if (getToken(1).kind == LBRACE){  
 break;  
 }  
 else if (getToken(1).kind == EOF)  
 {  
 break;  
 }  
  
 t = getNextToken();  
 }  
  
 if (this.errors>= 10) {  
 for(Report r : this.syntacticErrors) {  
 System.out.println(r.toString());  
 }  
 throw new RuntimeException("Ten errors were found. Program terminated.");  
 }  
 };  
}  
  
PARSER\_END(Parser)  
  
SKIP :  
{  
 " " | "\r" | "\t" | "\n"  
}  
  
/\* COMMENTS \*/  
  
MORE :  
{  
 "//" : IN\_SINGLE\_LINE\_COMMENT  
|  
 <"/\*\*" ~["/"]> { input\_stream.backup(1); } : IN\_FORMAL\_COMMENT  
|  
 "/\*" : IN\_MULTI\_LINE\_COMMENT  
}  
  
<IN\_SINGLE\_LINE\_COMMENT>  
SPECIAL\_TOKEN :  
{  
 <SINGLE\_LINE\_COMMENT: "\n" | "\r" | "\r\n"> : DEFAULT  
}  
  
<IN\_FORMAL\_COMMENT>  
SPECIAL\_TOKEN :  
{  
 <FORMAL\_COMMENT: "\*/" > : DEFAULT  
}  
  
<IN\_MULTI\_LINE\_COMMENT>  
SPECIAL\_TOKEN :  
{  
 <MULTI\_LINE\_COMMENT: "\*/" > : DEFAULT  
}  
  
<IN\_SINGLE\_LINE\_COMMENT,IN\_FORMAL\_COMMENT,IN\_MULTI\_LINE\_COMMENT>  
MORE :  
{  
 < ~[] >  
}  
  
TOKEN:  
{  
 <IMPORT: "import">  
 | <STATIC: "static">  
 | <DOT: ".">  
 | <STAR: "\*">  
 | <SEMICOLON: ";">  
 | <COMMA: ",">  
 | <CLASS: "class">  
 | <EXTENDS: "extends">  
 | <LBRACE: "{">  
 | <RBRACE: "}">  
 | <PUBLIC: "public">  
 | <LPAREN: "(">  
 | <RPAREN: ")">  
 | <RETURN: "return">  
 | <VOID: "void">  
 | <MAIN: "main">  
 | <STRINGARR: "String[]">  
 | <INT: "int">  
 | <BOOLEAN: "boolean">  
 | <IF: "if">  
 | <ELSE: "else">  
 | <WHILE: "while">  
 | <ASSIGN: "=">  
 | <LBRACKET: "[">  
 | <RBRACKET: "]">  
 | <SC\_AND: "&&">  
 | <LESS: "<">  
 | <PLUS: "+">  
 | <MINUS: "-">  
 | <SLASH: "/">  
 | <TRUE: "true">  
 | <FALSE: "false">  
 | <THIS: "this">  
 | <NEW: "new">  
 | <BANG: "!">  
 | <LENGTH: "length">  
 | <IDENTIFIER: ["a"-"z", "A"-"Z", "\_", "$"](["a"-"z", "A"-"Z", "0"-"9", "\_", "$"])\* >  
 | <NUMERIC: (["0"-"9"])+ >  
}  
  
SimpleNode Program() : {}  
{  
 ( ImportDeclaration() )\* ClassDeclaration() <EOF> {  
  
 if (errors>= 1) {  
 for(Report r : syntacticErrors) {  
 System.out.println(r.toString());  
 }  
 }  
 return jjtThis;  
 }  
}  
  
  
void ImportDeclaration() : {}  
{  
 <IMPORT> Name() <SEMICOLON>  
}  
  
/\*  
 \* A lookahead of 2 is required below since "Name" can be followed  
 \* by a ".\*" when used in the context of an "ImportDeclaration".  
 \*/  
void Name() : {}  
{  
 Identifier() (LOOKAHEAD(2) <DOT> Identifier() )\*  
}  
  
void ClassDeclaration() : {}  
{  
 <CLASS>  
 Identifier()  
 (<EXTENDS> Identifier())?  
 <LBRACE>  
 (VarDeclaration() <SEMICOLON>)\*  
 (MethodDeclaration())\*  
 <RBRACE>  
}  
  
void VarDeclaration() : {}  
{  
 LOOKAHEAD(2)  
 Type() Identifier()  
 |  
 Expression()  
}  
  
void MethodDeclaration() : {}  
{  
 <PUBLIC>  
 (Method() | Main())  
}  
  
void Identifier() : { }  
{  
 <IDENTIFIER>   
}  
  
void Method() : {}  
{  
 Type()  
 Identifier()  
 <LPAREN>  
 (  
 Args()  
 )?  
 <RPAREN>  
 <LBRACE>  
 MethodDeclarationBody()  
 <RETURN> Expression() <SEMICOLON>  
 <RBRACE>  
}  
  
void Args() : {}  
{  
 VarDeclaration()  
 (  
 <COMMA> VarDeclaration()  
 )\*  
}  
  
void Main() : {}  
{  
 <STATIC>  
 <VOID>  
 <MAIN>  
 <LPAREN>  
 <STRINGARR>  
 Identifier()  
 <RPAREN>  
 <LBRACE>  
 MethodDeclarationBody()  
 <RBRACE>  
}  
  
void MethodDeclarationBody() : {}  
{  
 (BlockStatement())\*  
}  
  
void BlockStatement() : {}  
{  
 LOOKAHEAD(2)  
 LocalVariableDeclaration()  
 |  
 Statement()  
}  
  
void LocalVariableDeclaration() : {}  
{  
 Type() VariableDeclarator() (<COMMA> VariableDeclarator())\* <SEMICOLON>  
}  
  
void Type() : {}  
{  
 Identifier()  
 |  
 <BOOLEAN>   
 |  
 LOOKAHEAD(2)  
 <INT> <LBRACKET> <RBRACKET>   
 |  
 <INT>   
  
}  
  
void VariableDeclarator() : {}  
{  
 VariableDeclaratorId() [<ASSIGN> VariableInitializer()]  
}  
  
void VariableDeclaratorId() : {Token t;}  
{  
 Identifier() (<LBRACKET> <RBRACKET>)\*  
}  
  
void VariableInitializer() : {}  
{  
 ArrayInitializer()  
 |  
 Expression()  
}  
  
void ArrayInitializer() : {}  
{  
 <LBRACE> [VariableInitializer() (LOOKAHEAD(2) <COMMA> VariableInitializer())\*][<COMMA>] <RBRACE>  
}  
  
void Expression() : {}  
{  
 ConditionalAndExpression()  
}  
  
void ConditionalAndExpression() : {}  
{  
 RelationalExpression() (<SC\_AND> RelationalExpression())\*  
}  
  
void RelationalExpression() : {}  
{  
 AdditiveExpression() (<LESS> AdditiveExpression())\*  
}  
  
void AdditiveExpression() : {}  
{  
 SubtrativeExpression() (<PLUS> SubtrativeExpression())\*  
}  
  
void SubtrativeExpression() : {}  
{  
 MultiplicativeExpression() (<MINUS> MultiplicativeExpression())\*  
}  
  
void MultiplicativeExpression() : {}  
{  
 DivisionExpression() (<STAR> DivisionExpression())\*  
}  
  
void DivisionExpression() : {}  
{  
 UnaryExpressionNotPlusMinus() (<SLASH> UnaryExpressionNotPlusMinus())\*  
}  
  
void UnaryExpressionNotPlusMinus() : {}  
{  
 <BANG> UnaryExpressionNotPlusMinus()  
 |  
 PrimaryExpression()  
}  
  
void PrimaryExpression() : {}  
{  
 PrimaryPrefix() (PrimarySuffix())?  
}  
  
void PrimaryPrefix() : {}  
{  
 Literal()  
 |  
 <THIS>   
 |  
 <LPAREN> Expression() <RPAREN>  
 |  
 AllocationExpression()   
}  
  
void PrimarySuffix() : {}  
{  
 InsideArray()  
 |  
 DotExpression()  
}  
  
/\*void DotExpression() : {}  
{  
 <DOT>  
 (  
 (<LENGTH>)  
 |  
 (Identifier() <LPAREN> (Args())? <RPAREN>)  
 )  
}\*/  
  
void DotExpression() : {}  
{  
 <DOT>  
 (  
 <LENGTH>  
 |  
 (Identifier() <LPAREN> (Expression() (<COMMA> Expression())\*)? <RPAREN>)  
 )  
}  
  
void Literal() : {}  
{  
 <NUMERIC>  
 |  
 Identifier()  
 |  
 BooleanLiteral()  
}  
  
void BooleanLiteral() : {}  
{  
 <TRUE>  
 |  
 <FALSE>  
}  
  
void Arguments() : {}  
{  
 <LPAREN> [ArgumentList()] <RPAREN>  
}  
  
void ArgumentList() : {}  
{  
 Expression() (<COMMA> Expression())\*  
}  
  
void AllocationExpression() : {}  
{  
 <NEW>  
 (  
 <INT> InsideArray()   
 |  
 Identifier() <LPAREN> <RPAREN>  
 )  
}  
  
void InsideArray() : {}  
{  
 <LBRACKET> Expression() <RBRACKET>  
}  
  
void Statement() : {}  
{  
 IfStatement()  
 |  
 WhileStatement()  
 |  
 ( LOOKAHEAD(2)  
 IdentifierStatement() | Expression() ) <SEMICOLON>  
 |  
 <LBRACE> (Statement())\* <RBRACE>  
}  
  
void IfStatement() : {}  
{  
 <IF> <LPAREN> (Expression()) <RPAREN> (Statement())  
 <ELSE> (Statement())  
}  
  
void WhileStatement() : {}  
{  
 <WHILE> WhileCondition() Statement()  
}  
  
void WhileCondition() : {}  
{  
 try {  
 <LPAREN> Expression() <RPAREN>}  
 catch (ParseException e)  
 {  
 errorFunction(e);  
 }  
}  
  
void IdentifierStatement() : {}   
{  
 (Identifier() ( InsideArray() ) ?) <ASSIGN> Expression()  
}