The Jack Compiler 1:
Syntax Analysis

TokenType
Enum

- KEYWORD
- SYMBOL
- INT_CONST
- STRING_CONST
- IDENTIFIER

Kind
Enum

- FIELD
- STATIC
- LOCAL
- ARGUMENT

Keyword
Enum

- CLASS
- METHOD
- FUNCTION
- CONSTRUCTOR
- INT
- BOOLEAN
- CHAR
- VOID
- VAR
- STATIC
- FIELD
- LET
- DO
- IF
- ELSE
- WHILE
- RETURN
- TRUE
- FALSE
- NULL
- THIS
- toString

JackTokenizer
Class

Fields
- cleanInput : string
- inputFile : Scanner
- intVal : int
- keyword : Keyword
- stringVal : string
- symbol : char
- tokenType : TokenType

Methods
- advance() : void
- getIdentifier() : string
- getIntVal() : int
- getKeyword() : Keyword
- getStringVal() : string
- getSymbol() : char
- getTokenType() : TokenType
- hasMoreTokens() : bool
- isKeyword() : bool
- JackTokenizer(string fileName)
- main() : void
- printElement(string tag, string value, PrintWriter outputFile) : void
- readIdentifier() : void
- readIntConst() : void
- readStringConst() : void
- readSymbol() : void

CompilationEngine
Class

Fields
- className : string
- labelNumber : int
- outputFile : PrintWriter
- symbolTable : HashSet<string, string>
- tokenizer : JackTokenizer

Methods
- CompilationEngine(string inputFileNa...
- compileClass() : void
- compileClassVarDec() : void
- compileDo() : void
- compileExpression() : void
- compileExpressionList() : void
- compileIf() : void
- compileLet() : void
- compileParameterList() : void
- compileReturn() : void
- compileStatements() : void
- compileSubroutine() : void
- compileTerm() : void
- compileVarDec() : void
- compileWhile() : void

JackAnalyzer
Class

Methods
- main() : void

Jack tokens

keyword: 'class'|'constructor'|'function'|
'method'|'field'|'static'|'var'|'int'|
'char'|'boolean'|'void'|'true'|'false'|
'null'|'this'|'let'|'do'|'if'|'else'|
'while'|'return'

symbol: '{'|'}'|'('|')'|'['|']'|'.'|','|';'|'+'|'-'|'*'|
'/'|'&'|'|'|'<'|'>'|'='|'~'

integerConstant: a decimal number in the range 0 … 32767

StringConstant: '"' a sequence of Unicode characters,
not including double quote or newline '"'

identifier: a sequence of letters, digits, and
underscore ('_') not starting with a digit.

Jack tokenizer

TestClass.jack
```
...
if (x < 0) {
    let sign = "negative";
}
...
```
→ tokenizer →

TestClassT.xml
```
<tokens>
    <keyword> if </keyword>
    <symbol> ( </symbol>
    <identifier> x </identifier>
    <symbol> &lt; </symbol>
    <integerConstant> 0 </integerConstant>
    <symbol> ) </symbol>
    <symbol> { </symbol>
    <keyword> let </keyword>
    <identifier> sign </identifier>
    <symbol> = </symbol>
    <stringConstant> negative </stringConstant>
    <symbol> ; </symbol>
    <symbol> } </symbol>
</tokens>
```

string constants are outputted
without the double-quotes

<, >, ", and & are outputted as
<, >, ", and &

JackTokenizer API

JackTokenizer: Ignores all comments and white space in the input stream, and serializes it into Jack-language tokens. The token types are specified according to the Jack grammar.

Routine	Arguments	Returns	Function
Constructor	input file / stream		Opens the input .jack file and gets ready to tokenize it.
hasMoreTokens	—	boolean	Are there more tokens in the input?
advance	—		Gets the next token from the input, and makes it the current token. This method should be called only if hasMoreTokens is true. Initially there is no current token.
tokenType	—	KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST	Returns the type of the current token, as a constant.

keyWord	—	CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS	Returns the keyword which is the current token, as a constant. This method should be called only if tokenType is KEYWORD.
symbol	—	char	Returns the character which is the current token. Should be called only if tokenType is SYMBOL.
identifier	—	string	Returns the identifier which is the current token. Should be called only if tokenType is IDENTIFIER.
intVal	—	int	Returns the integer value of the current token. Should be called only if tokenType is INT_CONST.
stringVal	—	string	Returns the string value of the current token, without the two enclosing double quotes. Should be called only if tokenType is STRING_CONST.

The Jack Language Grammar

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	`'class'` | `'constructor'` | `'function'` | `'method'` | `'field'` | `'static'` | `'var'` | `'int'` | `'char'` | `'boolean'` | `'void'` | `'true'` | `'false'` | `'null'` | `'this'` | `'let'` | `'do'` | `'if'` | `'else'` | `'while'` | `'return'`
symbol:	`'{'` | `'}'` | `'('` | `')'` | `'['` | `']'` | `'.'` | `','` | `';'` | `'+'` | `'-'` | `'*'` | `'/'` | `'&'` | `'|'` | `'<'` | `'>'` | `'='` | `'~'`
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	`'"'` A sequence of Unicode characters not including double quote or newline `'"'`
identifier:	A sequence of letters, digits, and underscore (`'_'`) not starting with a digit.
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	`'class'` className `'{'` classVarDec* subroutineDec* `'}'`
classVarDec:	(`'static'` | `'field'`) type varName (`','` varName)* `';'`
type:	`'int'` | `'char'` | `'boolean'` | className
subroutineDec:	(`'constructor'` | `'function'` | `'method'`) (`'void'` | type) subroutineName `'('` parameterList `')'` subroutineBody
parameterList:	((type varName) (`','` type varName)*)?
subroutineBody:	`'{'` varDec* statements `'}'`
varDec:	`'var'` type varName (`','` varName)* `';'`
className:	identifier
subroutineName:	identifier
varName:	identifier

The Jack Language Grammar (continued)

Statements:

statements:	statement*
statement:	letStatement | ifStatement | whileStatement | doStatement | returnStatement
letStatement:	`'let'` varName (`'['` expression `']'`)? `'='` expression `';'`
ifStatement:	`'if'` `'('` expression `')'` `'{'` statements `'}'` (`'else'` `'{'` statements `'}'`)?
whileStatement:	`'while'` `'('` expression `')'` `'{'` statements `'}'`
doStatement:	`'do'` subroutineCall `';'`
ReturnStatement	`'return'` expression? `';'`

Expressions:

expression:	term (op term)*
term:	integerConstant | stringConstant | keywordConstant | varName | varName `'['` expression `']'` | subroutineCall | `'('` expression `')'` | unaryOp term
subroutineCall:	subroutineName `'('` expressionList `')'` | (className | varName) `'.'` subroutineName `'('` expressionList `')'`
expressionList:	(expression (`','` expression)*)?
op:	`'+'` | `'-'` | `'*'` | `'/'` | `'&'` | `'|'` | `'<'` | `'>'` | `'='`
unaryOp:	`'-'` | `'~'`
KeywordConstant:	`'true'` | `'false'` | `'null'` | `'this'`