Jack

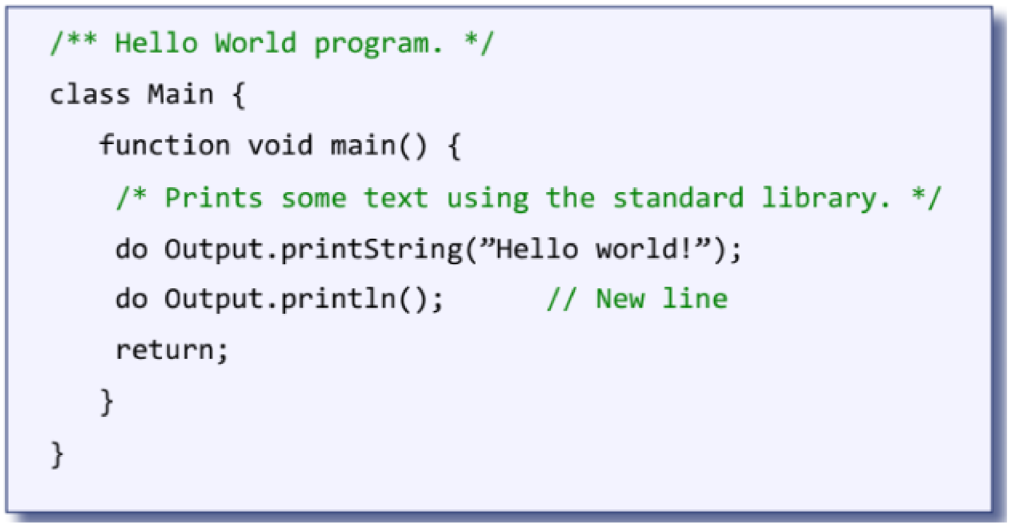
- Comments:

/** API block comment */

/* block comment */

// in-line comment

- White space (ignored)



```
/** Hello World program. */
class Main {
   function void main() {
    /* Prints some text using the standard library. */
    do Output.printString(”Hello world!”);
    do Output.println();      // New line
    return;
   }
}
```

- A Jack program is a collection of one or more Jack classes, one of which must be named Main
- The Main class must have at least one function, named main
- Program’s entry point: Main.main

Jack data types:

Primitive:

- int
- char
- boolean

Class types:

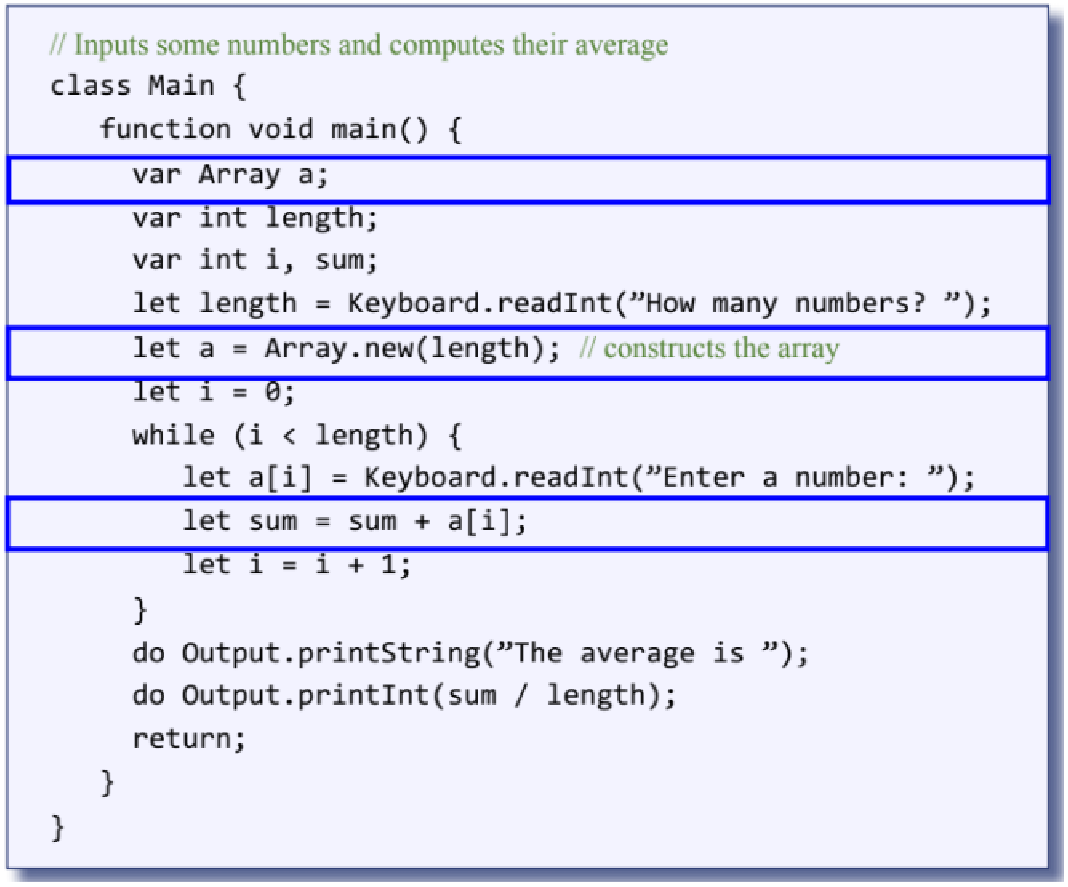
- OS: Array, String, ...
- Additional ADT’s can be defined and used, as needed

Flow of control:

- if / if...else
- while
- do

Arrays:

- Array is implemented as part of the standard class library
- Jack arrays are not typed



```
// Inputs some numbers and computes their average
class Main {
   function void main() {
      var Array a;
      var int length;
      var int i, sum;
      let length = Keyboard.readInt(”How many numbers? ”);
      let a = Array.new(length); // constructs the array
      let i = 0;
      while (i < length) {
         let a[i] = Keyboard.readInt(”Enter a number: ”);
         let sum = sum + a[i];
         let i = i + 1;
      }
      do Output.printString(”The average is ”);
      do Output.printInt(sum / length);
      return;
   }
}
```

OS services:

- Keyboard.readInt
- Output.printString
- Output.printInt
- More...

the only way to access field values from outside the class is through accessors

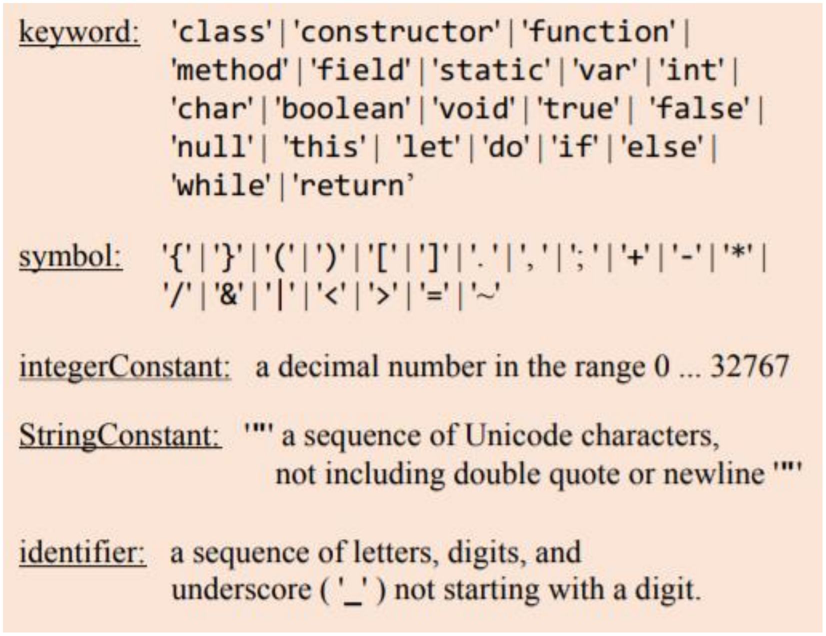
Jack subroutines:

- methods
- constructors
- functions

- this: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object
- a subroutine must terminate with a return command

Syntax elements:

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers



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

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

<u>integerConstant:</u> a decimal number in the range 0 ... 32767

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

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

Primitive types

- int: Non-negative 2's-complement 16-bit integer, i.e. an integer in the range 0,..., 32767
- boolean: true or false
- char: Integer values representing characters

Class types

- OS types: String, Array
- User-defined types: Fraction, List, ...

Jack subroutines

- Constructors: create new objects
- Methods: operate on the current object
- Functions: static methods

Subroutine types and return values

- Method and function type can be either void, a primitive data type, or a class name
- Each subroutine must end with return value or return.

Constructors

- 0, 1, or more in a class
- Common name: new
- The constructor's type must be the name of the constructor's class
- The constructor must return a reference to an object of the class type.

Variable kinds:

- field variables:

object properties, can be manipulated by the class constructors and methods

- static variables:

class-level variables, can be manipulated by the class subroutines

- local variables: used by subroutines, for local computations
- parameter variables:

used to pass values to subroutines, behave like local variables

Variables must be ...

- Declared before they are used
- Typed.

Statement	**Syntax**	**Description**
`let`	`let` *varName* = *expression*`;` or `let` *varName*`[`*expression*1`]` = *expression*2`;`	An assignment operation (where *varName* is either single-valued or an array). The variable kind may be *static*, *local*, *field*, or *parameter*.

`if`	`if (`*expression*`) {`
	*statements*1
	`}`
	`else {`
	*statements*2
	`}`
		Typical *if* statement with an optional *else* clause.
		The curly brackets are mandatory even if *statements* is a single statement.

`while`	`while (`*expression*`) {`
	statements
	`}`
		Typical *while* statement.
		The curly brackets are mandatory even if *statements* is a single statement.

`do`	`do` *function-or-method-call*`;`	Used to call a function or a method for its effect, ignoring the returned value.

`return`	`Return` *expression*`;`
	or
	`return;`
		Used to return a value from a subroutine.
		The second form must be used by functions and methods that return a void value. Constructors must return the expression `this`.

A *Jack expression* is one of the following:

- A *constant*
- A *variable name* in scope. The variable may be *static*, *field*, *local*, or *parameter*
- The `this` keyword, denoting the current object (cannot be used in functions)
- An *array element* using the syntax *Arr*`[`*expression*`]`,
  where *Arr* is a variable name of type `Array` in scope
- A *subroutine call* that returns a non-void type
- An expression prefixed by one of the unary operators `-` or `~`:
  `-` *expression*: arithmetic negation
  `~` *expression*: boolean negation (bit-wise for integers)
- An expression of the form *expression op expression*
  where *op* is one of the following binary operators:

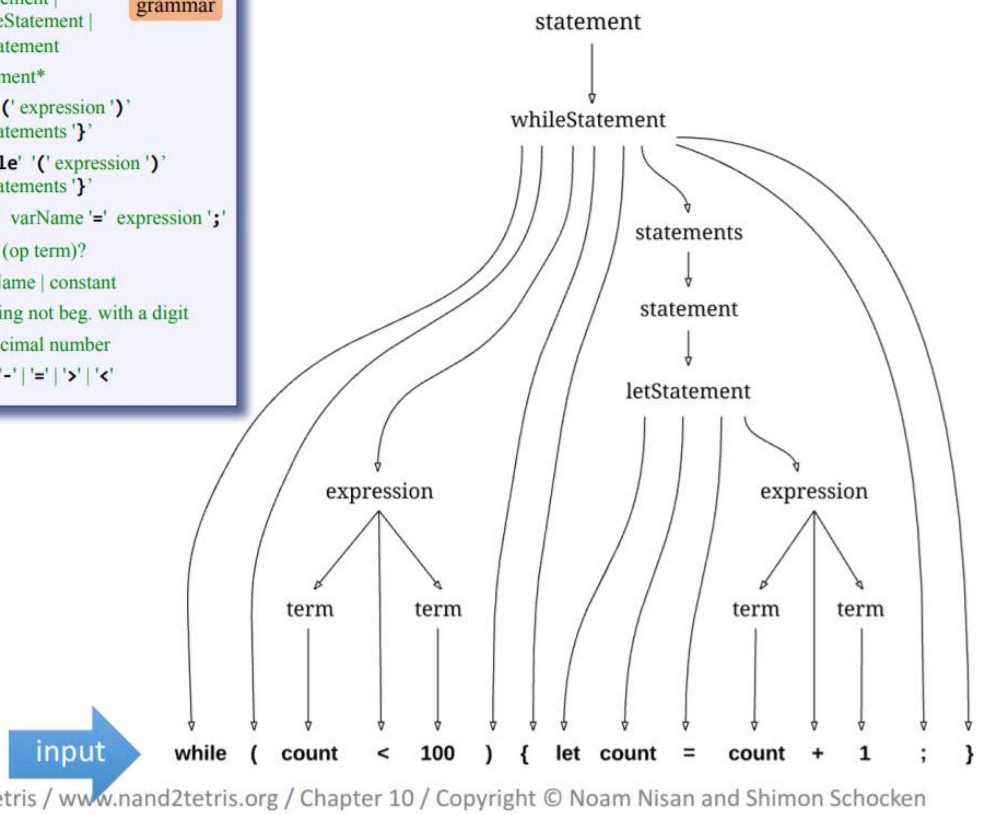
`+ - * /`	Integer arithmetic operators
`& |`	Boolean And and Boolean Or (bit-wise for integers) operators
`< > =`	Comparison operators

- `(`*expression*`)`: An expression in parenthesis

statement:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	'**`if`**' '**`(`**' expression '**`)`**' '**`{`**' statements '**`}`**'
whileStatement:	'**`while`**' '**`(`**' expression '**`)`**' '**`{`**' statements '**`}`**'
letStatement:	'**`let`**' varName '**`=`**' expression '**`;`**'
expression:	term (op term)?
term:	varName | constant
varName:	a string not beginning with a digit
constant:	a decimal number
op:	'**`+`**' | '**`-`**' | '**`=`**' | '**`>`**' | '**`<`**'

grammar

statement:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	'**if**' '(' expression ')' '{' statements '}'
whileStatement:	'**while**' '(' expression ')' '{' statements '}'
letStatement:	'**let**' varName '=' expression ';'
expression:	term (op term)?
term:	varName | constant
varName:	a string not beg. with a digit
constant:	a decimal number
op:	'+' | '-' | '=' | '>' | '<'



grammar

statement:	ifStatement | whileStatement | letStatement
statements:	statement*
ifStatement:	**'if'** '(' expression ')' '{' statements '}'
whileStatement:	**'while'** '(' expression ')' '{' statements '}'
letStatement:	**'let'** varName '=' expression ';'
expression:	term (op term)?
term:	varName | constant
varName:	a string not beg. with a digit
constant:	a decimal number
op:	'+' | '-' | '=' | '>' | '<'

parser output

```
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term>
         <identifier> count </identifier>
      </term>
      <symbol> < </symbol>
      <term>
         <intConstant> 100 </intConstant>
      </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <letStatement>
         <keyword> let </keyword>
         <identifier> count </identifier>
         <symbol> = </symbol>
         <expression>
            <term> <identifier> count </identifier> </term>
            <symbol> + </symbol>
            <term> <intConstant> 1 </intConstant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
```

Same parse tree, in XML

If the parser encounters a *terminalElement* *xxx* of type
keyword, *symbol*, *integer constant*, *string constant*, or *identifier*,

the parser generates the output:

```
<terminalElement>
   xxx
</terminalElement>
```

where *terminalElement* is:
`keyword`,
`symbol`,
`integerConstant`,
`stringConstant`,
`identifier`

Examples:

```
<keyword> method </keyword>

<symbol> { </symbol>

<integerConstant> 42 </integerConstant>

<stringConstant> xkcd </stringConstant>

<symbol> { </symbol>
```

the parser generates the output:

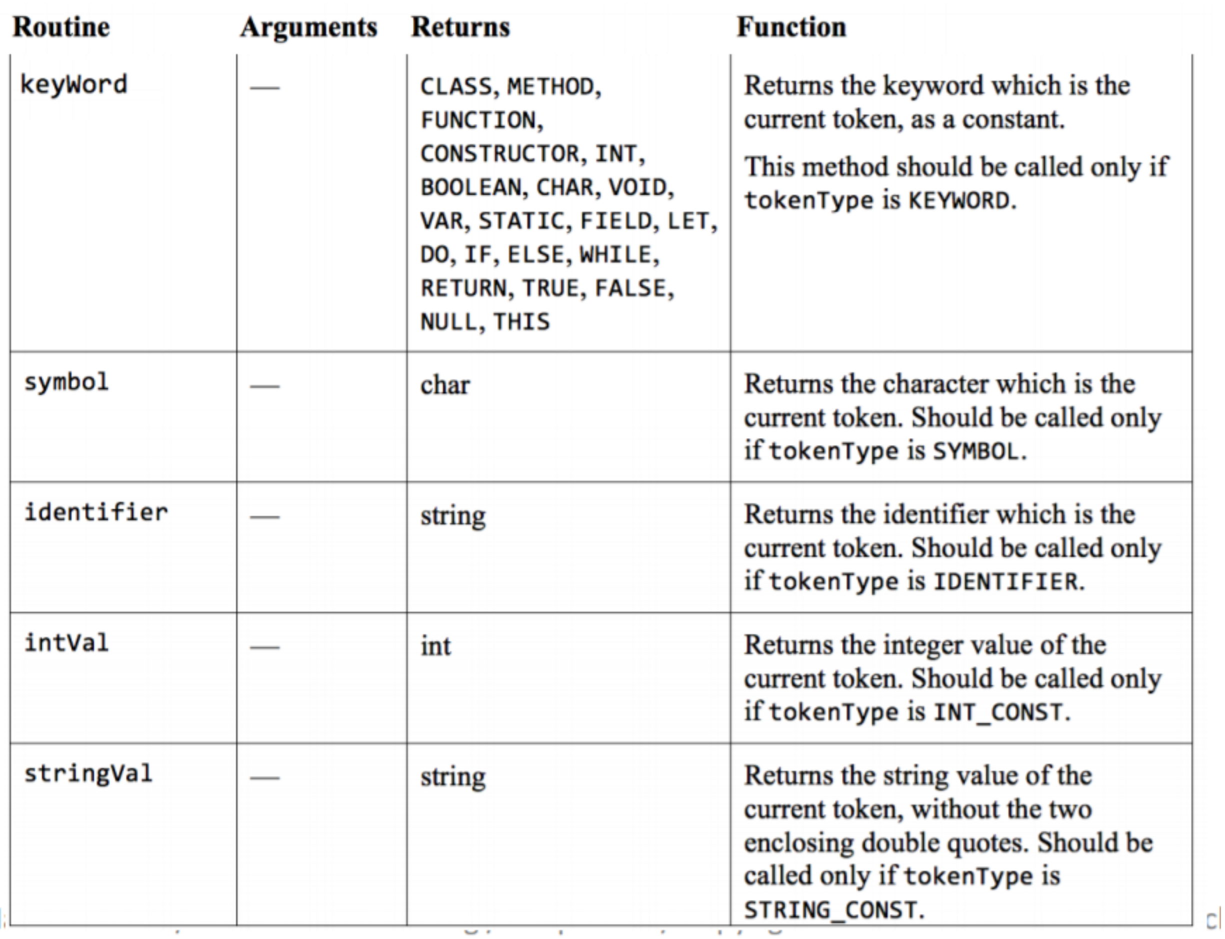
```
<nonTerminal>
    Recursive output for the non-terminal body
</nonTerminal>
```

where *nonTerminal* is:
`class`, `classVarDec`, `subroutineDec`, `parameterList`, `subroutineBody`, `varDec`; `statements`, `LetStatement`, `ifStatement`, `whileStatement`, `doStatement`, `returnStatement`; `expression`, `term`, `expressionList`

Example: if the input is `return x;`

```
<returnStatement>
   <keyword>
      return
   </keyword>
   <expression>
      <term>
         <identifier> x </identifier>
      </term>
   </expression>
   <symbol> ; </symbol>
</returnStatement>
```

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.



Routine	Arguments	Returns	Function
`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`.

Routine	Arguments	Returns	Function
Constructor	Input stream/file Output stream/file		Creates a new compilation engine with the given input and output. The next routine called must be `compileClass`.
`CompileClass`	—	—	Compiles a complete class.
`CompileClassVarDec`	—	—	Compiles a static variable declaration, or a field declaration.
`CompileSubroutineDec`	—	—	Compiles a complete method, function, or constructor.
`compileParameterList`	—	—	Compiles a (possibly empty) parameter list. Does not handle the enclosing "`()`".
`compileSubroutineBody`	—	—	Compiles a subroutine's body.
`compileVarDec`	—	—	Compiles a `var` declaration.
`compileStatements`	—	—	Compiles a sequence of statements. Does not handle the enclosing "`{}`".

CompilationEngine API

`CompilationEngine`: generates the compiler's output.

Routine	Arguments	Returns	Function
`compileLet`	—	—	Compiles a `let` statement.
`compileIf`	—	—	Compiles an `if` statement, possibly with a trailing `else` clause.
`compileWhile`	—	—	Compiles a `while` statement.
`compileDo`	—	—	Compiles a `do` statement.
`compileReturn`	—	—	Compiles a `return` statement.

Routine	Arguments	Returns	Function
`CompileExpression`	--	--	Compiles an expression.
`CompileTerm`	--	--	Compiles a *term*. If the current token is an *identifier*, the routine must distinguish between a *variable*, an *array entry*, or a *subroutine call*. A single look-ahead token, which may be one of "`[`", "`(`", or "`.`", suffices to distinguish between the possibilities. Any other token is not part of this term and should not be advanced over.
`CompileExpressionList`	--	--	Compiles a (possibly empty) comma-separated list of expressions.