Lab #07 Assembler

Name___________________________

Convert the following assembly code into "symbol less" code by replacing each symbol (variable or label) with its corresponding value (number). Also, please label the ROM address (line number) for each real instruction.

1. Sum.asm

Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)
// Computes sum = R2 + R3 // (R2 refers to RAM[2]) @R2 D=M @R3 D=D+M // Add R2 + R3 @sum M=D // sum = R2 + R3		

2. Max.asm

Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)

// Computes R2=max(R0, R1)
// (R0,R1,R2 refer to
// RAM[0],RAM[1],RAM[2])
 	
   @R0
   D=M
   @R1
   D=D-M
   @OUTPUT_FIRST
   D;JGT
   @R1
   D=M
   @OUTPUT_D
   0;JMP
(OUTPUT_FIRST)
   @R0
   D=M
(OUTPUT_D)
   @R2
   M=D
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP		

Lab #07 Assembler

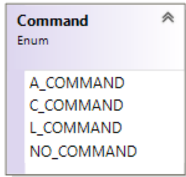
3. Rect.asm

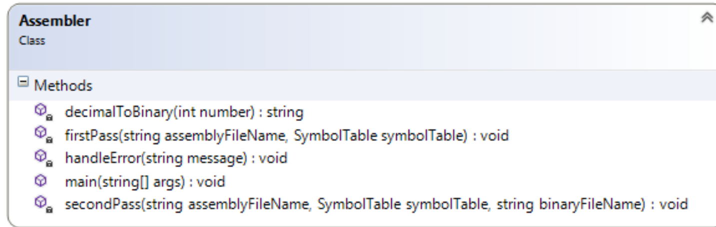
Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)

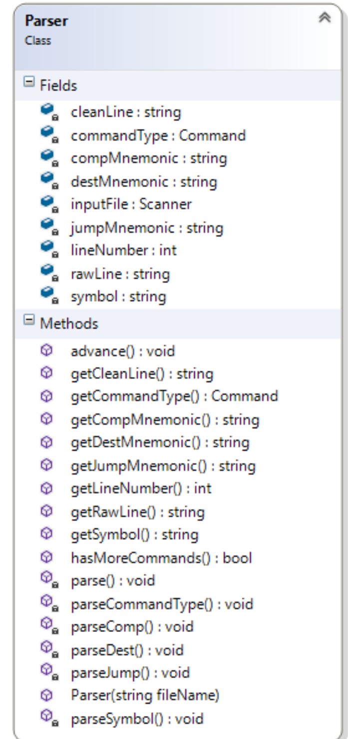
// Draws a rectangle at
// the top-left corner of
// the screen.
// The rectangle is 16
// pixels wide and R0
// pixels high.
 	
   @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @address
   M=D
(LOOP)
   @address
   A=M
   M=-1
   @address
   D=M
   @32
   D=D+A
   @address
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP		

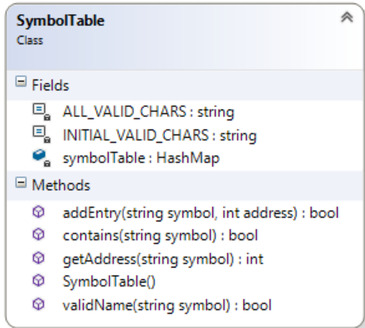
Lab #07 Assembler

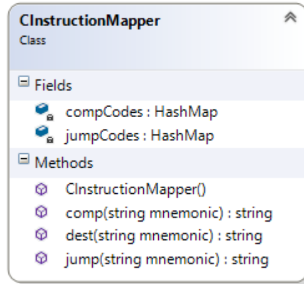
UML Diagram of Entire Assembler Program











A brief Java refresher:

Write Java code to implement the following enum:

Command
Enum

A_COMMAND
C_COMMAND
L_COMMAND
NO_COMMAND

What does the following code display?

```
String code = "\t0;JMP  //unconditional jump  ";
System.out.println(code.trim());
```

How would you extract the JMP from the code string above? Write the Java code to do so.

What is assigned to the variable dest ?

```
String code = "D=M;JGT";
int index = code.indexOf('=');
String dest = (index != -1) ?
      code.substring(0, index) : null;
```

Lab #07 Assembler

Write pseudocode for the following helper methods:

- ☐ String cleanLine(String rawLine)

```
//DESCRIPTION:   cleans raw instruction by removing non-essential parts
//PRECONDITION:  String parameter given (not null)
//POSTCONDITION: returned without comments and whitespace
```

- ☐ Command parseCommandType(String cleanLine)

```
//DESCRIPTION:   determines command type from parameter
//PRECONDITION:  String parameter is clean instruction
//POSTCONDITION: returns A_COMMAND (A-instruction),
//               C_COMMAND (C-instruction), L_COMMAND (Label) or
//               NO_COMMAND (no command)
```

Lab #07 Assembler

- [ ] boolean isValidName(String symbol)

```
//DESCRIPTION:   checks validity of identifiers for assembly code symbols
//PRECONDITION:  start with letters or "_.$:" only, numbers allowed after
//POSTCONDITION: returns true if valid identifier, false otherwise
```

- [ ] String decimalToBinary(int number)

```
//DESCRIPTION:   converts integer from decimal notation to binary notation
//PRECONDITION:  number is valid size for architecture, non-negative
//POSTCONDITION: returns 16-bit string of binary digits (first char is MSB)
```

Lab #07 Assembler

```
CInstructionMapper
------------------------------------------------
-  compCodes : HashMap<String, String>
-  destCodes : HashMap<String, String>
-  jumpCodes : HashMap<String, String>
------------------------------------------------
+  CInstructionMapper()
+  comp(mnemonic : String) : String
+  dest(mnemonic : String) : String
+  jump(mnemonic : String) : String
```

Write pseudocode for the following Code methods:

❑ Code()

```
//DESCRIPTION:   initializes hashmaps with binary codes for easy lookup
//PRECONDITION:  comp codes = 7 bits (includes a), dest/jump codes = 3 bits
//POSTCONDITION: all hashmaps have lookups for valid codes
```

❑ String comp(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (7) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null
```

Lab #07 Assembler

- ❑ String dest(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (3) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null
```

- ❑ String jump(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (3) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null
```

Lab #07 Assembler

Write pseudocode for the following SymbolTable methods:

SymbolTable
- INITIAL_VALID_CHARS : String - ALL_VALID_CHARS : String - symbolTable : HashMap<String, Integer>
+ SymbolTable() + addEntry(symbol : String, address : int) : boolean + contains(symbol : String) : boolean + getAddress(symbol : String) : int - isValidName(symbol : String) : boolean

- ❑ SymbolTable()

```
//DESCRIPTION:   initializes hashmap with predefined symbols
//PRECONDITION:  follows symbols/values from book/appendix
//POSTCONDITION: all hashmap values have valid address integer
```

- ❑ boolean addEntry(String symbol, int address)

```
//DESCRIPTION:   adds new pair of symbol/address to hashmap
//PRECONDITION:  symbol/address pair not in hashmap (check contains() 1st)
//POSTCONDITION: adds pair, returns true if added, false if illegal name
```

- ❑ boolean contains(String symbol)

```
//DESCRIPTION:   returns boolean of whether hashmap has symbol or not
//PRECONDITION:  table has been initialized
//POSTCONDITION: returns boolean if arg is in table or not
```

- ❑ int getAddress(String symbol)

```
//DESCRIPTION:   returns address in hashmap of given symbol
//PRECONDITION:  symbol is in hashmap (check w/ contains() first)
//POSTCONDITION: returns address associated with symbol in hashmap
```

- ❑ boolean isValidName(String symbol) //same as earlier but rewrite using constants

Lab #07 Assembler

```
Parser
------------------------------------------------------------------
+  NO_COMMAND : char // 'N'              //constants
+  A_COMMAND : char //  'A'
+  C_COMMAND : char //  'C'
+  L_COMMAND : char //  'L'

-  inputFile : Scanner                   //file stuff +
   debugging
-  lineNumber : int
-  rawLine : String

-  cleanLine : String              //parsed command parts
-  commandType : char
-  symbol : String
-  destMnemonic : String
-  compMnemonic : String
-  jumpMnemonic : String
------------------------------------------------------------------
+  Parser(inFileName : String)           //drivers
+  hasMoreCommands() : boolean
+  advance() : void

-  cleanLine() : void                    //parsing helpers
-  parseCommandType() : void
-  parse() : void
-  parseSymbol() : void
-  parseDest() : void
-  parseComp() : void
-  parseJump() : void

+  getCommandType() : char               //useful getters
+  getSymbol() : String
+  getDest() : String
+  getComp() : String
+  getJump() : String

+  getRawLine() : String                 //debugging getters
+  getCleanLine() : String
+  getLineNumber() : int
```

- ❑ cleanLine() : void //same as part 1 but rewrite using instance variables
- ❑ parseCommandType() : void //same as part 1 but rewrite using instance variables

Lab #07 Assembler

Write pseudocode for the following Parser methods:

- ❑ Parser(String fileName)

```
//DESCRIPTION:   opens input file/stream and prepares to parse
//PRECONDITION:  provided file is ASM file
//POSTCONDITION: if file can't be opened, ends program w/ error message
```

- ❑ boolean hasMoreCommands()

```
//DESCRIPTION:   returns boolean if more commands left, closes stream if not
//PRECONDITION:  file stream is open
//POSTCONDITION: returns true if more commands, else closes stream
```

- ❑ void advance()

```
//DESCRIPTION:   reads next line from file and parses it into instance vars
//PRECONDITION:  file stream is open, called only if hasMoreCommands()
//POSTCONDITION: current instruction parts put into instance vars
```

Lab #07 Assembler

- ❑ void parseSymbol()

```
//DESCRIPTION:   parses symbol for A- or L-commands
//PRECONDITION:  advance() called so cleanLine has value,
//   call for A- and L-commands only
//POSTCONDITION: symbol has appropriate value from instruction assigned
```

- ❑ void parseDest()

```
//DESCRIPTION:   helper method parses line to get dest part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: destMnemonic set to appropriate value from instruction
```

- ❑ void parseComp()

```
//DESCRIPTION:   helper method parses line to get comp part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: compMnemonic set to appropriate value from instruction
```

Lab #07 Assembler

- ❑ void parseJump()

```
//DESCRIPTION:   helper method parses line to get jump part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: jumpMnemonic set to appropriate value from instruction
```

- ❑ void parse()

```
//DESCRIPTION:   helper method parses line depending on instruction type
//PRECONDITION:  advance() called so cleanLine has value
//POSTCONDITION: appropriate parts (instance vars) of instruction filled
```

Lab #07 Assembler

- [ ] Command getCommandType()

```
//DESCRIPTION:   getter for command type
//PRECONDITION:  cleanLine has been parsed (advance was called)
//POSTCONDITION: returns Command for type (N/A/C/L)
```

- [ ] String getSymbol()

```
//DESCRIPTION:   getter for symbol name
//PRECONDITION:  cleanLine has been parsed (advance was called),
//   call for labels only (use getCommandType())
//POSTCONDITION: returns string for symbol name
```

- [ ] String getDestMnemonic()

```
//DESCRIPTION:   getter for dest part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for dest part
```

- [ ] String getCompMnemonic()

```
//DESCRIPTION:   getter for comp part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for comp part
```

Lab #07 Assembler

- [ ] String getJumpMnemonic()

```
//DESCRIPTION:   getter for jump part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for jump part
```

- [ ] String getRawLine()

```
//DESCRIPTION:   getter for rawLine from file (debugging)
//PRECONDITION:  advance() was called to put value from file in here
//POSTCONDITION: returns string of current original line from file
```

- [ ] String getCleanLine()

```
//DESCRIPTION:   getter for cleanLine from file (debugging)
//PRECONDITION:  advance() and cleanLine() were called
//POSTCONDITION: returns string of current clean instruction from file
```

- [ ] int getLineNumber()

```
//DESCRIPTION:   getter for lineNumber (debugging)
//PRECONDITION:  n/a
//POSTCONDITION: returns line number currently being processed from file
```