CS224 – Jack Code Generation

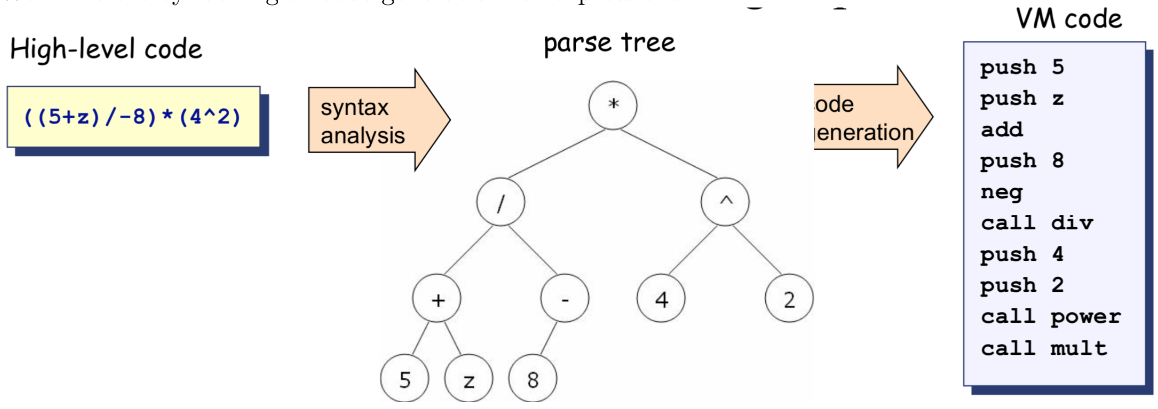
Purpose: The last phase of our Jack compiler is actually generating VM code. We will examine some code templates that will help you do that.

Knowledge: This activity will help you become familiar with the following content knowledge:

- What VM code needs to be generated for various scenarios

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 60 minutes.

1. We will start by looking at code generation for expressions:

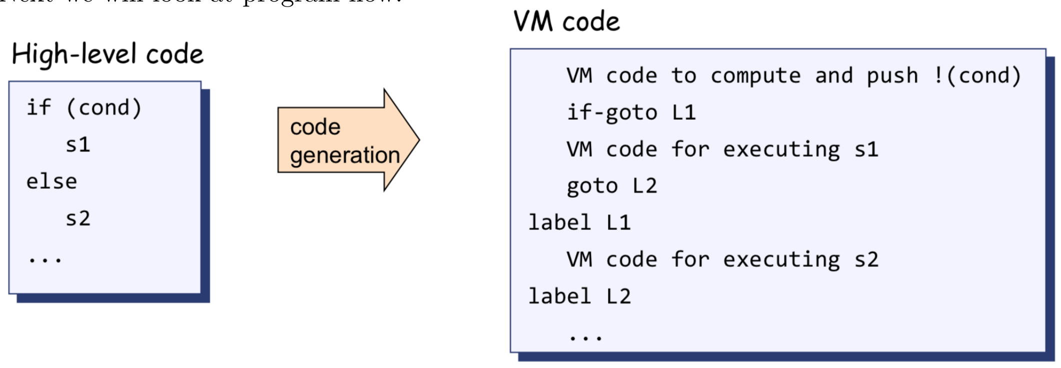


```
push 5
push z
add
push 8
neg
call div
push 4
push 2
call power
call mult
```

Complete the logic for generating VM code from a parse tree `exp` .

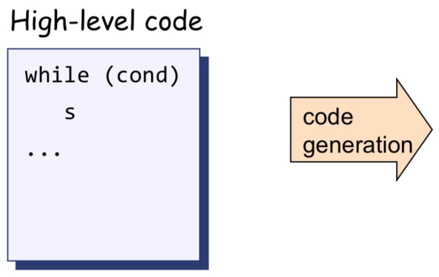
The `codeWrite(exp)` algorithm:
if `exp` is a constant `n` then output `push n`
if `exp` is a variable `v` then output ______________________
if `exp` is `op(exp1)` then ______________________________
if `exp` is `(exp1 op exp2)` then ______________________________
if `exp` is `f(exp1, ... expn)` then ______________________________

2. Next we will look at program flow:



```
if (cond)
   s1
else
   s2
...
```

```
   VM code to compute and push !(cond)
   if-goto L1
   VM code for executing s1
   goto L2
label L1
   VM code for executing s2
label L2
   ...
```



```
while (cond)
   s
...
```

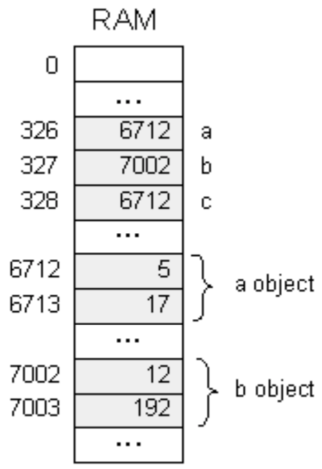
Give the VM code for a while loop.

3. Next we will consider how to handle objects. Consider the following classes:

Java code

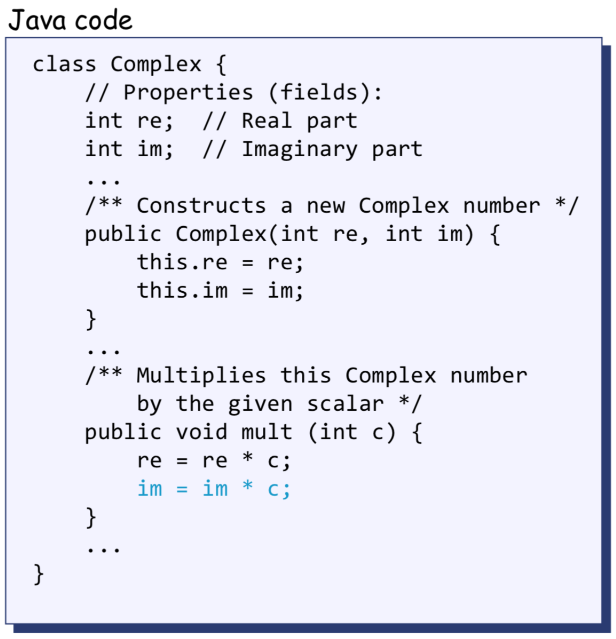
```
class Complex {
    // Fields (properties):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex (int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    public void bla() {
        Complex a, b, c;
        ...
        a = new Complex(5,17);
        b = new Complex(12,192);
        ...
        c = a; // Only the reference is copied
        ...
  }
```

After compilation the RAM might look like this:



So what tasks must the generated code perform when constructing an object like `x = new Classname(...)` ?

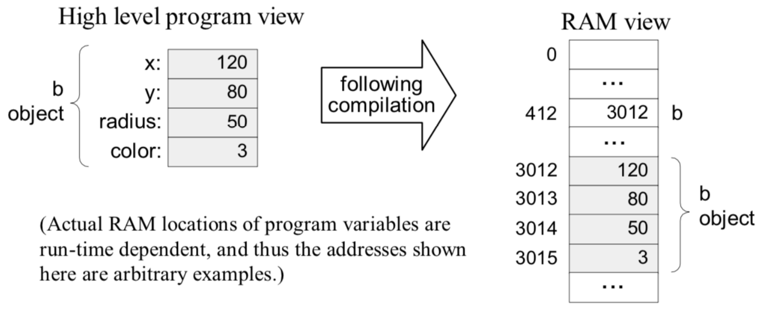
4. What would be the VM code for the assignment statement `im = im * c`?
Hint: We would look up the two variables in the symbol table and `im` would be in segment `this` with and index of 1 and `c` would be in segment `arg` with an index of 0



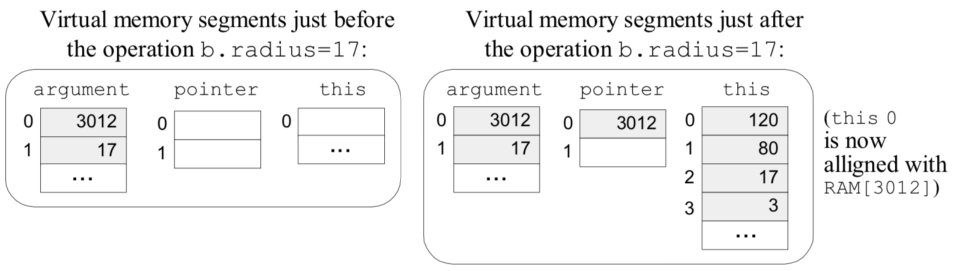
Java code

```
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
    /** Multiplies this Complex number
        by the given scalar */
    public void mult (int c) {
        re = re * c;
        im = im * c;
    }
    ...
}
```

5. Next we will consider accessing the fields in an object. Suppose we have an object named `b` of type `Ball`. A Ball has `x`,`y` coordinates, a `radius`, and a `color`.



If we execute `b.radius = 17`, we would set `pointer[0]` to the address of the object `b`. This would cause the `this` segment to align with the object `b` so that we can access the field.


Virtual memory segments just before the operation b.radius=17:
argument
0 3012
1 17
...
pointer
0
1
this
0 ...
Virtual memory segments just after the operation b.radius=17:
argument
0 3012
1 17
...
pointer
0 3012
1
this
0 120
1 80
2 17
3 3
...
(this 0 is now alligned with RAM[3012])

Suppose we have a function in which `b` and `r` are passed in as the first two arguments. Give the VM code for `b.radius = r`
Hint: Get `b`'s address and set `pointer[0]` to that value. Store `r`'s value in `b`'s third field which is now the third location in the `this` segment.

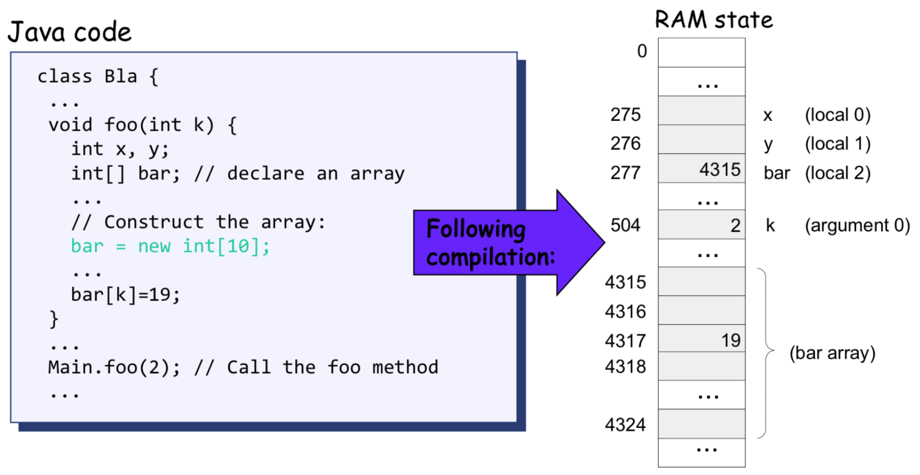
6. We will next look at method calls. A method call `x.mult(5)` can also be viewed as `mult(x,5)` since the first parameter is the object itself.

   Java code

```java
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    ...
    public void bla() {
        Complex x;
        ...
        x = new Complex(1,2);
        x.mult(5);
        ...
  }
}
```

   So what would the VM code for this method call look like?

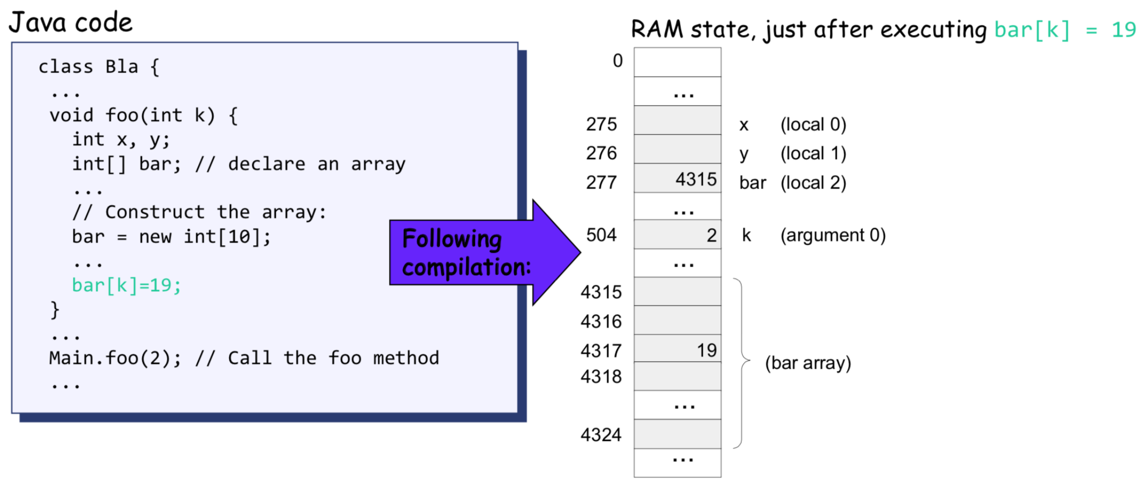
7. We can use the `memory.alloc(n)` library function which allocates n memory locations and returns the address of the start of this block. Consider creating an array:


Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

So what would the VM code be for `bar = new int(10)` ?

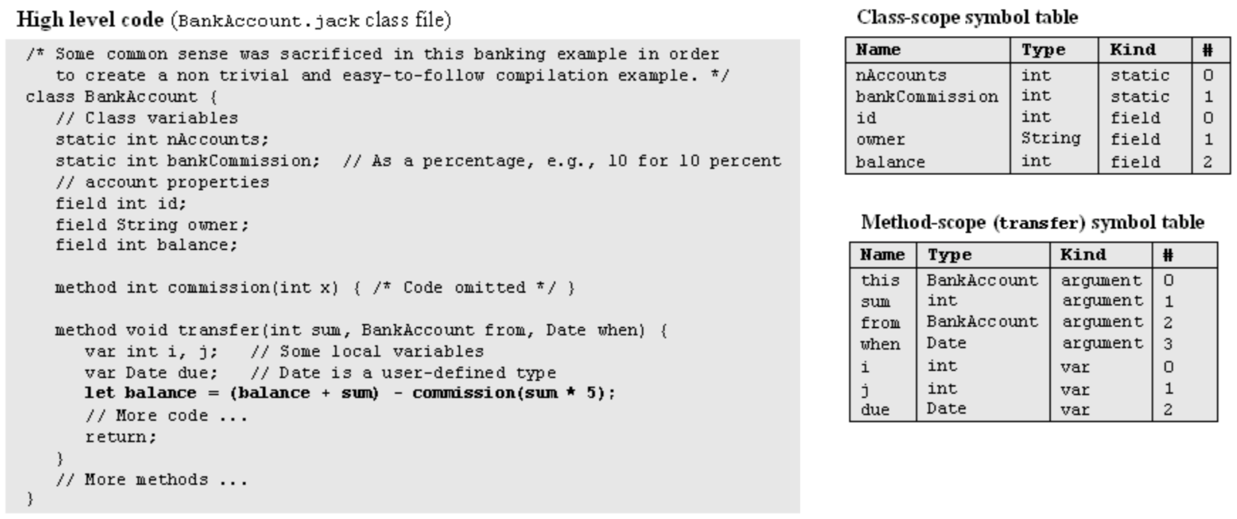
8. To access an array element by its index we can use the `that` segment. If we put the address of that element into `pointer[1]` then it will align the memory address with the `that` segment.


Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

Give the VM code for `bar[k]=19`

9. We will now write the VM code for method declarations. Consider the following example:



High level code (`BankAccount.jack` class file)

```
/* Some common sense was sacrificed in this banking example in order
   to create a non trivial and easy-to-follow compilation example. */
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;  // As a percentage, e.g., 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) { /* Code omitted */ }

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;   // Some local variables
      var Date due;   // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
      return;
   }
   // More methods ...
}
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (`transfer`) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

The VM code uses `function` and `return` to define a method. All methods return a value and can be 0 if no value is actually needed to be returned. The `function` takes the name for the function and the number of local variables as parameters. To call a function we first push the arguments and then use `call` with the name of the function and the number of parameters for that function as parameters for the call.

Complete the VM code below:

```
function BankAccount.commission 0
// Code omitted
function BankAccount.transfer 3
push argument 0
pop pointer 0
<Your code here>
// More code
push 0
return
```

10. We will examine the VM code for expressions. Explain why the following VM code represents each of these types of expressions:

    (a) Assuming you have the `varSeg` and `varIndex` from the symbol table, explain the VM code for the expression `varName`

    ```
    push varSeg varIndex
    ```

    (b) Assuming you have the `className` of the file being compiled and you can recursively create the VM code for the expression list, explain the VM code for `subName(expressionList)`
    Hint: The arg0 is `this`

    ```
    push pointer 0
    <compile expressionList>
    call className.subName <nArgs>
    ```

    What we would do in CompilationEngine.java to get the number of arguments used in the last statment?

    (c) Assuming you have the `varSeg`, `varIndex`, and `varType` in the symbol table, and can recursively create the VM code for the expression list, explain the VM code for `varName.methodName(expressionList)`
    Hint: The arg0 is the object being acted upon. Also the `varType` is needed to know what class the method is defined in.

    ```
    push varSeg varIndex
    <compile expressionList>
    call varType.methodName <nArgs>
    ```

    (d) Explain the VM code for `className.methodName(expressionList)`

    ```
    <compile expressionList>
    call className.methodName <nArgs>
    ```

    How would we distinguish that we have a className rather than a varName when we are compiling this code?
    Where in the CompilationEngine.java code would we get the `className`?

    (e) Assuming you have the `varSeg` and `varIndex` from the symbol table, explain the VM code for `varName[expression]`
    Hint: `pointer[1]` aligns the `that` segment to the array.

    ```
    <compile expression>
    push varSeg varIndex
    add
    pop pointer 1
    push that 0
    ```

11. Look at the CompilationEngine.java code that I gave in the Parser starter code and answer the following questions.

    (a) Not all of our parser methods generate VM code. Which methods do not?

    (b) We need to generate unique labels in `compileIfStatement` and `compileWhileStatement`. How can we generate these unique label names?

    (c) Unique code needs to be generated at the beginning of `compileSubroutineBody` if we are compiling a constructor or compiling a method.

        i. What will the unique code for constructors do?

        ii. What will the unique code for methods do?

        iii. How would `compileSubroutineBody` determine the type value for the subroutine. Where is that information generated?

    (d) The code generation for a string constant in `compileTerm` involves two steps. First you have to call `String.new` to create a string object. Then you need to repeatedly call `String.appendChar` to add all the characters to that string object. What would be the parameter to the `String.new` call?

    (e) What code would be generated for each of the terms: `true`, `false`, `null`, `this` ?

    It looks like you are ready for the code generation project now!!