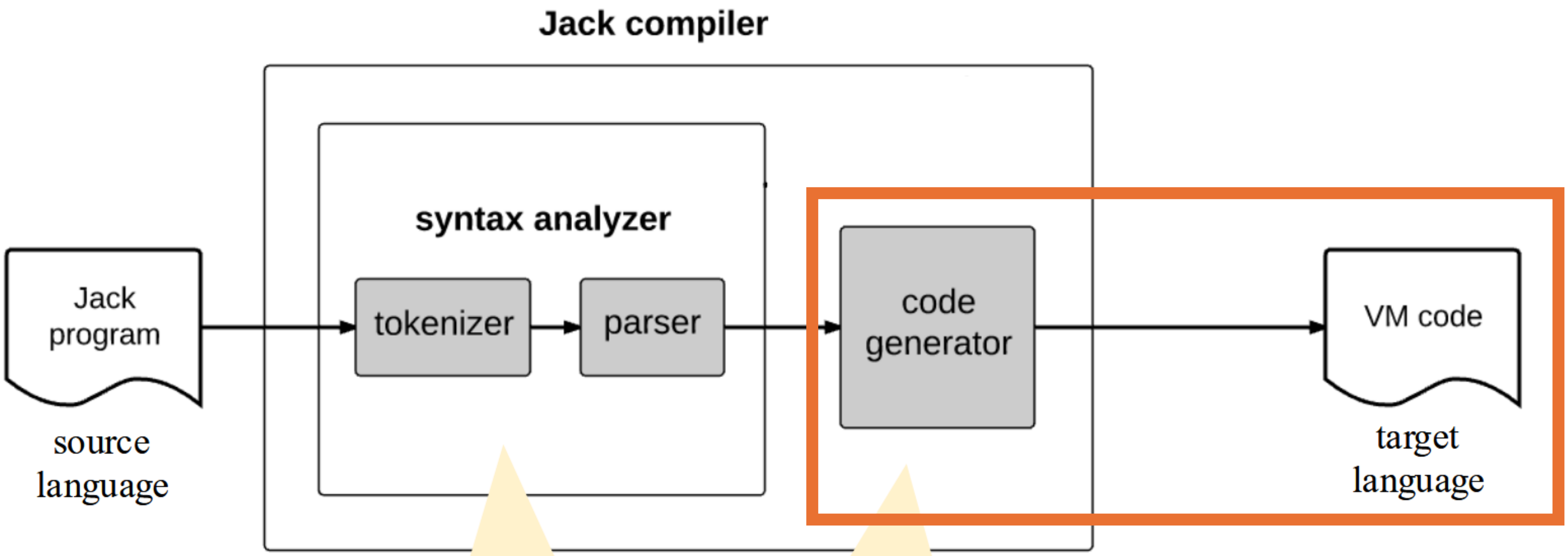
Agenda

Compilation Part 2: From parsing to VM code generation

- Variables (Symbol Tables)
- Expressions (Expression trees)
- Statements
- Objects, constructors, methods

(with slides from nand2tetris)

Compiler Roadmap



Understanding / parsing the semantics of the *source code*

Expressing the parsed semantics using the *target language*

Variables and symbol tables

A **symbol table** associates variable names with their attributes

name: identifier

type: int, char, Boolean, class name

kind: field, static, local, argument

scope: subroutine level, class level, global

For Jack, we will maintain two symbol tables

- class-level symbol table
- method-level symbol table

Example: Variables and symbol table

```
class Point {
  field int x, y;
  static int pointCount;
  ...
  method int distance(Point other) {
     var int dx, dy;
     let dx = x - other.getx();
     let dy = y - other.gety();
     return Math.sqrt((dx*dx) + (dy*dy));
  }
  ...
}
```

What variables are defined in this program?
What is their scope? class or method
What is their type, kind, and number?

class-level

Name	type	kind	#

method-level

Name	type	kind	#

Nested scoping

Nested scoping refers to the ability to have different depths of variable scope.

```
class foo {
    // class-level variable declarations
    method bar () {
        // method-level variable declarations
        ...
            {
                // scope-1-level variable declarations
                ...
                    {
                        // scope-2-level variable declarations
                        ...
```

static variables: Persist throughout the program's execution

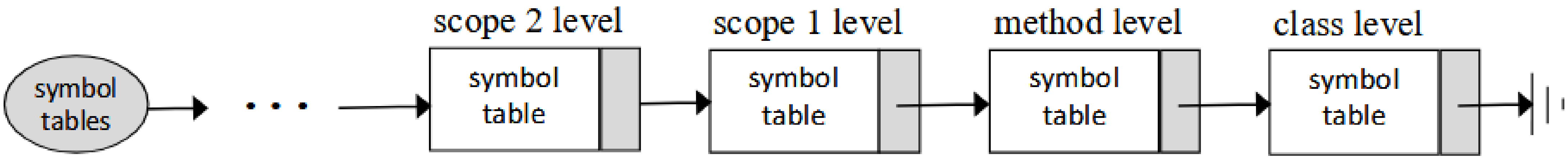
Field variables: Persist as long as the object is not disposed

Local variables: Each time a subroutine starts running during runtime, it gets a fresh set of local variables; Each time a subroutine returns, its local variables are recycled

Argument variables: Same as local variables

Nested scoping

<u>Nested scoping</u> can be handled using a linked list of symbol tables:



<u>Variable lookup</u>: The variable is looked up in the first table in the list: if not found, the next table is looked up, and so on.

Algorithm: Name resolution in statements

Compiling statements:

For each variable found in a statement:

The compiler looks up the variable in the method-level symbol table;

If found, the variable is replaced with its *segment i reference*;

Else, the compiler looks up the variable in the class-level symbol table;

If found, the variable is replaced with its *segment i reference*;

Else, the compiler throws a compilation error.

Example: Symbol table entry to virtual memory segment references

```
class Main {
  function int foo(int a, int b) {
    var int dx;
    let dx = a - b;
    return Math.sqrt((dx*dx));
  }
  ...
}
```

Name	type	kind	#

// Jack

let dx = a – b;

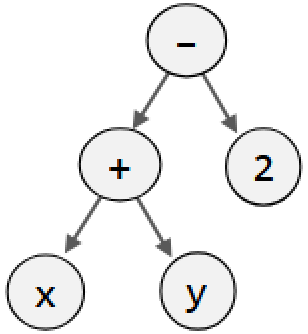
// VM

Compiling expressions

infix notation

`x + y - 2`

human friendly



postfix notation

`x y + 2 -`

stack automata friendly

generate

VM code

```
// x + y - 2
push x
push y
add
push 2
sub
```

Rule: *expression: term (op term)?*

Post-order traversal of the parse tree

When executed during runtime, the generated code ends up leaving the value of the expression at the stack's top

Example: Compiling expressions

Expression

foo(x, y+1, -7)

Parse Tree

VM Code

Compiling expressions: Code Generation

Expression compilation algorithm:

```
compileExpression(exp):

  if exp is term:
     compileTerm(term)

  if exp is "term1 op1 term2  op2
             term3 op3 ...  termn":
     compileTerm(term1)
     compileTerm(term2)
     output "op1"
     compileTerm(term3)
     output "op2"
     ...
     compileTerm(termn)
     output "opn-1"
```

```
compileTerm(term):

  if term is a constant c:
     output "push c"

  if term is a variable var:
     output "push var"

  if term is "unaryOp term":
     compileTerm(term)
     output "unaryOp"

  if term is "f(exp1, exp2, ...)":
     compileExpression(exp1)
     compileExpression(exp2)
     ...
     compileExpression(expn)
     output "call f n"

  if term is "(exp)":
     compileExpression(exp)
```

Generated code (examples)

```
// x + y - 2
push x
push y
add
push 2
sub
```

```
// foo(x, y+1,-7)
push x
push y
push 1
add
push 7
neg
call foo 3
```

From the Jack grammar:

term: *constant* | *varName* | *unaryOp term* | *subroutineCall* | (*expression*)

subroutineCall: f ($expression_1$, $expression_2$, … , $expression_n$)

expression: $term_1$ op_1 $term_2$ op_2 $term_3$ op_3 … $term_n$

Compiling statements: Let

source code (Jack)

```
...
let varName = expression;
...
```

compiler

VM code (generated by `compileLet`)

```
...
VM code generated by compileExpression
pop varName
...
```

example

```
let v = g + r2;
```

compiler

```
push g
push r2
add
pop v
```

1. first three commands are generated by `compileExpression`
2. Instead of symbolic variable names, the generated VM code uses segment entries.

compileLet:

`compileExpression`

output "`pop static / this / argument / local` *i* "

Compiling statements: Return

source code

```
...
return;
...
```

compiler →

VM code (generated by `compileReturn`)

```
...
push constant 0
return
...
```

`compileReturn`:

output "`push constant 0`"

output "`return`"

When compiling a `return` Jack statement with no return value, the `compileReturn` routine generates code that pushes a dummy value onto the stack;

The dummy value will be tossed away by the compiled code of the caller (discussed next).

Compiling statements: Do

source code (Jack)

```
...
do subroutineName(exp1, exp2, ...)
...
```

Used to call a function or a method for its effect, ignoring the returned value

compiler

VM code (generated by `compileDo`)

```
...
VM code generated by compileExpression
pop temp 0
...
```

The `pop` gets rid of the return value.

example

```
do Output.printInt(7);
```

compiler

```
push constant 7
call Output.printInt 1
pop temp 0
```

first two commands are generated by `compileExpression`

Compiling statements: If

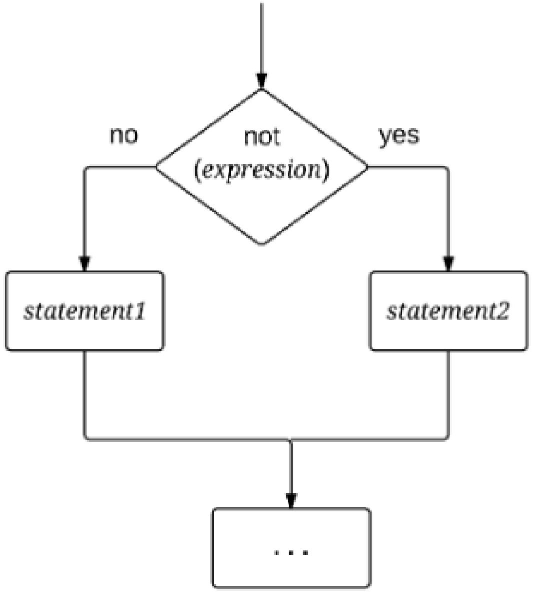
source code

```
if ( expression )
   { statements1 }
else
   { statements2 }
...
```

compiler

VM code (generated by `compileIf`)

```
   VM code generated by compileExpression
   not
   if-goto L1
   VM code generated by compileStatements
   goto L2
label L1
   VM code generated by compileStatements
label L2
   ...
```



`compileStatements`: a simple compilation routine that uses a loop to compile a sequence of zero or more `let`, `return`, `do`, `if`, `while` statements

Compiling statements: If code generation

<u>compileIf:</u>

compileExpression
output "not”
generate a unique label L1, and output "if-goto L1”
compileStatements
generate a unique label L2, and output "goto L2”
output "label L1”
compileStatements
output "label L2"

example

```
if (x < 0) {
   let y = 0;
} else {
   let y = 1;
}
...
```

compiler

```
   push x
   push 0
   lt
   not
if-goto L1
   push 0
   pop y
   goto L2
label l1
   push 1
   pop y
label L2
   ...
```

Compiling statements: While

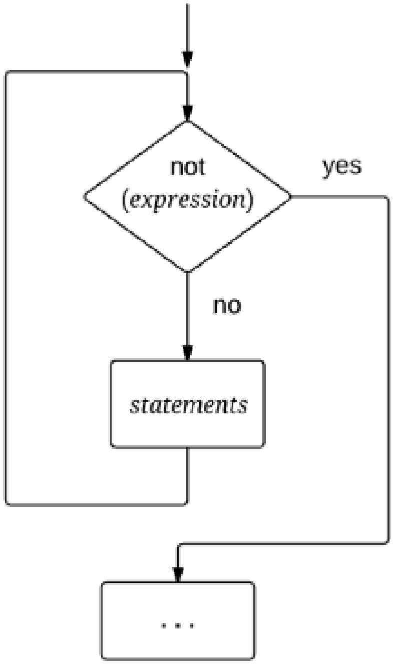
source code

```
while ( expression )
    { statements }
...
```

compiler

VM code (generated by `compileWhile`)

```
label L1
   VM code generated by compileExpression
   not
   if-goto L2
   VM code generated by compileStatements
   goto L1
label L2
   ...
```



Compiling statements: While code generation

<u>compileWhile:</u>

generate the unique label L1, and output "label L1"
compileExpression
output "not"
generate the unique label L2, and output "if-goto L2"
compileStatements
output "goto L1"
output "label L2"

example

```
while (g = 0) {
   let x = x + 1;
}
...
```

compiler

```
label L1
   push g
   push 0
   eq
   not
   if-goto L2
   push x
   push 1
   add
   pop x
   goto L1
label L2
   ...
```

Compiling functions: Function calls

source code

```
...
let y = Math.sqrt(x);
...
```

compiler

VM code

```
...
push local 0
call Math.sqrt 1
pop local 1
...
```

name	type	kind	#
x	int	local	0
y	int	local	1
...			

symbol table of the calling subroutine

<u>Nothing special about it</u>

Compiling *function calls* is part of *compiling expressions* (already discussed)

<u>Observation</u>

There's nothing special about compiling *function calls*;

They are handled as part of compiling *expressions*.

Compiling functions: function bodies

source code

```
...
let y = Math.sqrt(x);
...
```

source code

```
class Math {
  ...
  /** Square root */
  function int sqrt(int x) {
     var int g, prevg, epsilon;
     let epsilon = 1;
     let g = x / 2;     // initial guess
     while (true) {
        let prevg = g;
        let g = (g + x / g) / 2;
        if (abs(prevg - g) < epsilon) {
           return g;
        }
     }
  }
  ...
}
```

name	type	kind	#
x	int	argument	0
g	int	local	0
prevg	int	local	1
epsilon	int	local	2

subroutine-level symbol table

compiler

VM code

```
// class Math {
//// compileClass builds the class-level symbol table (omitted)
// function int sqrt(int x)
// var int g, prev, epsilon;
//// compileSubroutine calls compileParametersList and
//// compileVarDec, that build the subroutine's symbol table;
//// compileSoubroutine then generates VM code that declares
//// a VM function that has 3 local variables.
function Math.sqrt 3
    //// compileSoubroutine calls CompileStatements,
    //// that handles the method's body
    // let epsilon = 1;
   push constant 1
   pop local 2
   // let g = x / 2;
   push argument 0
   push constant 2
   call Math.divide 2
   ...
   push local 0
   return
```

Compiling objects: creating instances

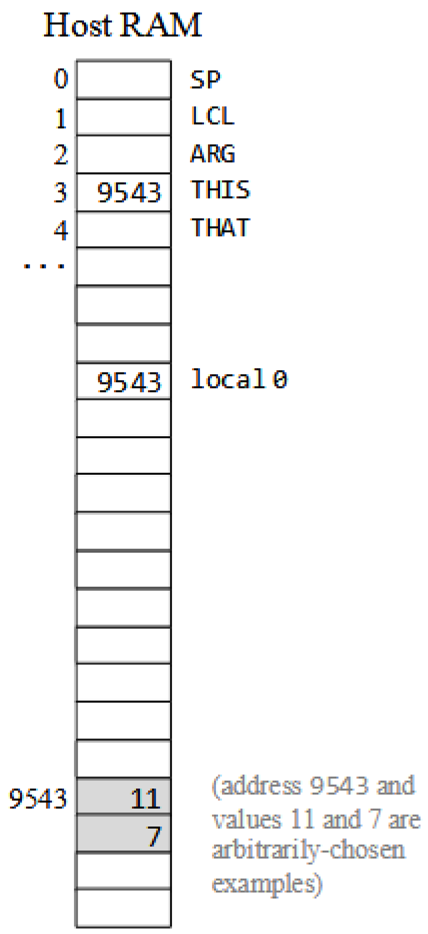
Recall: Objects store their data sequentially in RAM

Creating a structure:

```
// Creates a 2-word structure, initializes it, and makes local 0 refer to it
push constant 2
call Memory.alloc 1
pop pointer 0     // pop THIS

push constant 11
pop this 0
push constant 7
pop this 1

push pointer 0    // push THIS
pop local 0
```



Compiling objects: creating instances

Recall: Objects store their data sequentially in RAM

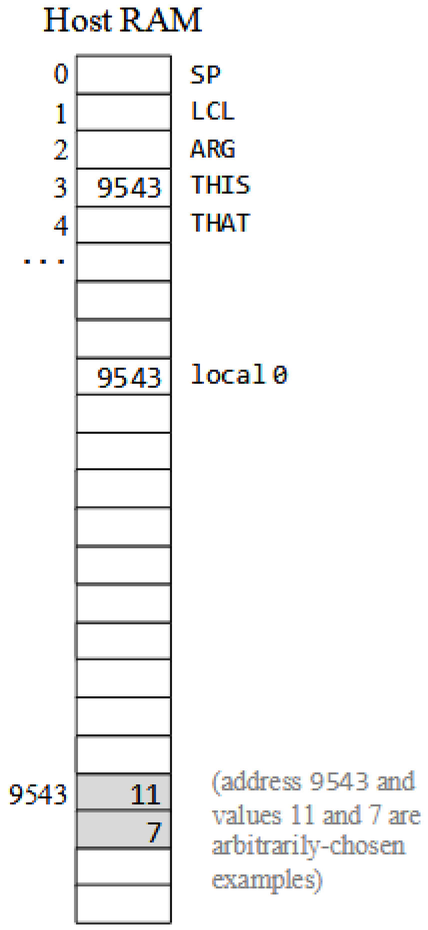
Creating a structure:

```
// Creates a 2-word structure, initializes it, and make
push constant 2
call Memory.alloc 1
pop pointer 0      // pop THIS
push constant 11
pop this 0
push constant 7
pop this 1
push pointer 0     // push THIS
pop local 0
```

Memory.alloc(n): Finds a memory block of size n words, and returns (pushes onto the stack) its base address

pop pointer 0: Pops the base address of the new structure into THIS; Result: Subsequent this i references will effect RAM addresses THIS + i

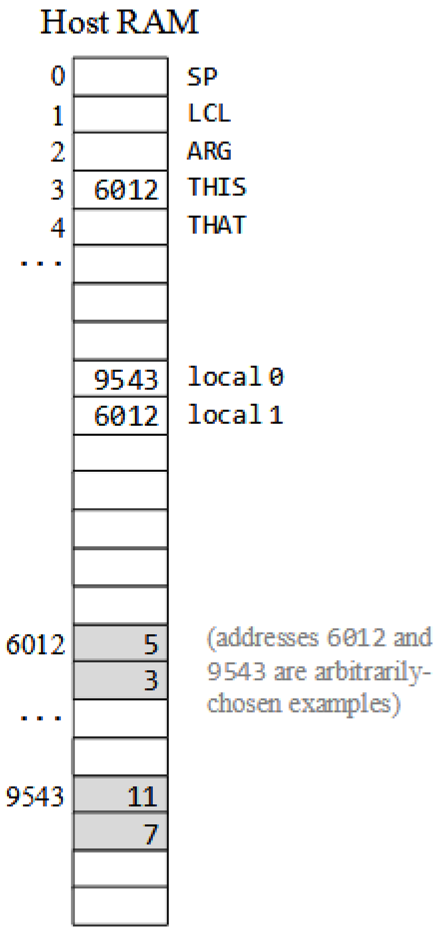
Makes local 0 point to the new structure



Compiling objects: getting/setting fields

Manipulating a structure:

```
// Creates two structures and makes local 0 and local 1 point to them
// (code omitted)
...
// Example of manipulating a structure:
// Sets the fields of the the structure that local 1 refers to
push local 1
pop pointer 0    // THIS = base address of the target structure
push constant 5
pop this 0
push constant 3
pop this 1
```



Note

Using this technique, we can get / set the fields of any given structure.

Compiling objects: constructor code generation

Source code

```
...
// Creates a new Point object
let p1 = Point.new(2,3);
```

```
/** Represents a Point. */
class Point {
   field int x, y;
   static int pointCount;
    ...
    /** Constructs a new point. */
    constructor Point new(int ax, int ay) {
       let x = ax;
       let y = ay;
       let pointCount = pointCount + 1;
       return this;   //// required in Jack constructors
    }
    ...
```

compiler

VM code (created by compileClass)

```
// class Point {
//    field int x, y;
//    static int pointCount;
//// compileClass builds the class-level symbol table
// constructor Point new(int ax, int ay);
//// compileSubroutine builds the subroutine's symbol table, and notes that
//// it is handling a constructor. It generates VM code that declares a VM function,
//// allocates memory for the new object, and sets THIS to its base address:
function Point.new 0
  push constant 2
  call Memory.alloc 1
  pop pointer 0
  //// CompileStatements handles the constructor's body
  // let x = ax;
  push argument 0
  pop this 0

  ...

  // return this;
  push pointer 0
  return
```

class-level symbol table

name	type	kind	#
x	int	this	0
y	int	this	1
pointCount	int	static	0

constructor-level symbol table

name	type	kind	#
ax	int	argument	0
ay	int	argument	1

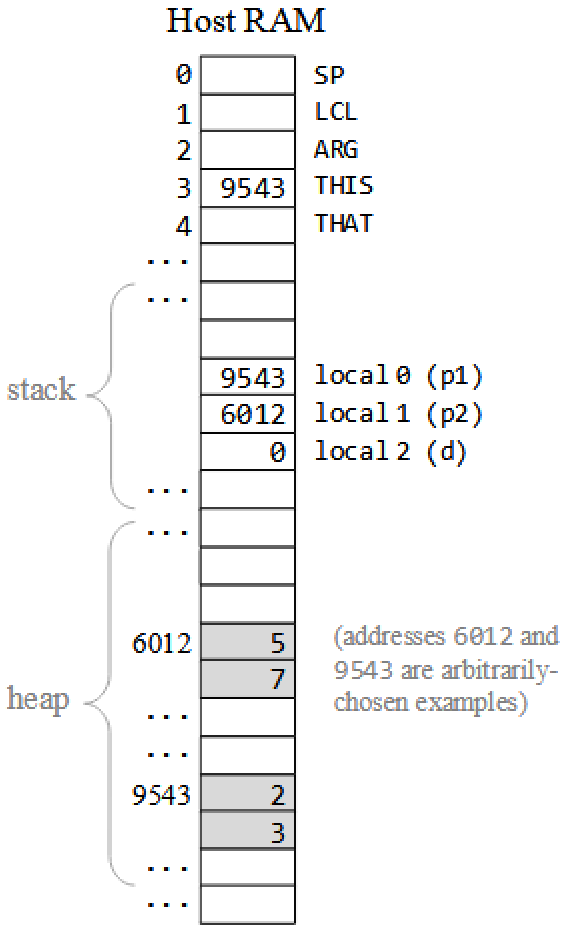
Compiling objects: methods

source code

```
...
let d = p1.distance(p2)
...
```

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Distance from this to the other point */
  method int distance(Point other) {
    var int dx, dy;
    let dx = x - other.getx();
    let dy = y - other.gety();
    return Math.sqrt((dx*dx) +
                     (dy*dy));
  }
  ...
}
```

We assume that the caller declared the local variables p1, p2, d, and constructed the objects p1 and p2 (code not shown)



Compiling objects: methods code generation

source code

```
...
let d = p1.distance(p2)
...
```

compiler

VM code

```
...
push local 0
push local 1
call Point.distance 2
pop local 2
...
```

name	type	kind	#
p1	Point	local	0
p2	Point	local	1
d	int	local	2

caller's symbol table

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Distance from this to the other point */
  method int distance(Point other) {
    var int dx, dy;
    let dx = x - other.getx();
    let dy = y - other.gety();
    return Math.sqrt((dx*dx) +
                     (dy*dy));
  }
  ...
}
```

name	type	kind	#
x	int	this	0
y	int	this	1
pointCount	int	static	0

class-level symbol table

name	type	kind	#
this	Point	argument	0
other	Point	argument	1
dx	int	local	0
dy	int	local	1

method-level symbol table

compiler

```
// class Point {
//   field int x, y;
//   static int pointCount;
//// compileClass builds the class-level symbol table
// method int distance(Point other)
// var int dx, dy;
//// compileSubroutine builds the subroutine's symbol table, and notes
//// that it is handling a method. It generates VM code that declares a VM
//// function and sets THIS to the object on which the method was called.
function Point.distance 2
   push argument 0
   pop pointer 0
   //// CompileStatements handles the method's body
   // let dx = x - other.getx();
   push this 0
   push argument 1
   call Point.getx 1
   sub
   pop local 0
   ...
   add
   call Math.sqrt 1
   return
```

Compiling arrays

Source code

```
// Can appear in any class/subroutine:
method foo() {
   // Declares an array variable:
   var Array arr;
   ...
   // Constructs the array:
   let arr = Array.new(5);
   ...
```

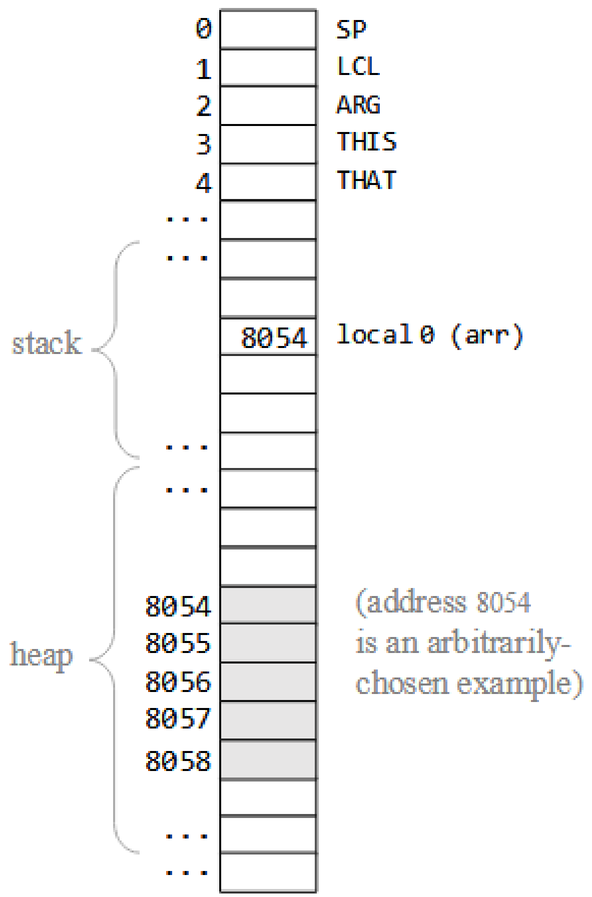
compile

VM code

```
// var Array arr;
//// compileSubroutine builds
//// the subrout.'s symbol table,
//// and adds to it variable arr.
...
// let arr = Array.new(5);
push constant 5
call Array.new 1
pop local 0
...
```

name	type	kind	#
arr	Array	local	0

Host RAM



<u>Compiling array construction:</u>

We simply compile the `let` statement

(same as with calling an object constructor)

Compiling arrays: Setting/getting values

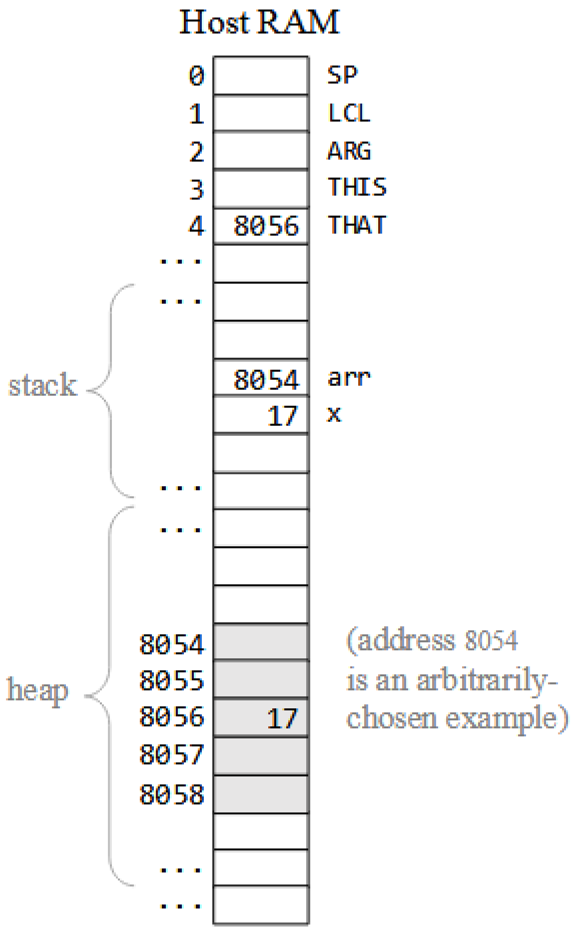
Examples

```
// let arr[2] = x
push arr
push 2
add
pop pointer 1
push x
pop that 0
```

Sets THAT to the address of the target array element

```
// let x = arr[2]
push arr
push 2
add
pop pointer 1
push that 0
pop x
```

same



Compiling arrays: working with expressions

source code

```
let arr[i] = exp
```

compiler

VM code (generated by `compileLet`)

```
push arr
push i
add
pop pointer 1    // THAT = arr + 2
push exp
pop that 0
```

Watch out! what about expressions like
let arr[i] = b[i];

Compiling arrays: working with expressions the right way

source code (example)

```
let a[i] = b[j]
```

compiler

VM code (genera

```
push a
push i
add
push b
push j
add
pop pointer 1     // THAT = address of b[j]
push that 0       // stack top = b[j]
pop temp 0        // temp 0 = b[j]
pop pointer 1     // THAT = address of a[i]
push temp 0       // stack top = b[j]
pop that 0        // a[i] = b[j]
```

Use temporary variables!

compileLet: (let *varName*[*expression*] = *expression*;)

```
// (both expressions can contain array elements, of any depth)
output "push varName"
call compileExpression
output "add"
call compileExpression
output "pop pointer 1"
...
output "pop that 0"
```

6 last pop/push commands at the VM code shown above

Implementation notes #1: Function names

Each subroutine (constructor, function, method) **subName** in a file **ClassName.jack** is compiled into a VM function named **className.subName**

Foo.jack

```
class Foo {

  constructor Foo new(int x) {}

  method void bar(int x) {}

  function int baz(int x) {}

}
```

JackCompiler

Foo.vm

```
function Foo.new
  ...
function Foo.bar
  ...
function Foo.baz
  ...
```

Implementation notes #2: constants

The Jack language has four constants

true is implemented in VM code as constant 1, followed by neg

false is implemented in VM code as constant 0

null is implemented in VM code as constant 0

this is implemented in VM code as pointer

Implementation notes #3: variables

Local variables

Are mapped on local 0, local 1, local 2, ...

Argument variables

Are mapped on argument 0, argument 1, argument 2, ...

Static variables

Are mapped on static 0, static 1, static 2, ...

Field variables

Are mapped on this 0, this 1, this 2, ...

Implementation notes #4: arrays

Access to array element arr[i] is implemented by generating VM code that realizes:

> *set pointer 1 to arr + i*
>
> *push / pop that 0*

Implementation tip: There is never a need to use *that i* for i greater than 0.

Implementation notes #5: subroutines

(Suppose that we are compiling the file ClassName.jack)

Compiling a constructor call or a function call subName(exp1, exp2, ..., expn)

1. The generated VM code pushes the expressions exp1, exp2, ..., expn onto the stack
2. Then call ClassName.subName n

Compiling a method call obj.subName(exp1, exp2, ..., expn)

1. The generated VM code pushes obj and then exp1, exp2, ..., expn onto the stack
2. Then call ClassName.subName n + 1

If the called subroutine is void

Just after the call, the generated VM code gets rid of the return value using the command pop temp

Implementation notes #5: subroutines (cont)

When compiling a Jack method:

- The first entry in the method's symbol table must be a variable named *this* whose type is the name of the class to which the method belongs, kind is argument, and index is 0
- The generated VM code starts by setting pointer 0 to argument 0

When compiling a Jack constructor:

- The generated VM code starts by calling the OS function Memory.alloc nFields (the number of fields in the class declaration)
- The return value must be *pointer 0*

When compiling a void function or a void method:

The generated VM code ends with push constant 0 and then return

Implementation notes #6: OS

The OS (here is the API) is organized as a set of 8 compiled Jack classes:

Math.vm, Memory.vm, Screen.vm, Output.vm, Keyboard.vm, String.vm, Array.vm, Sys.vm

Every OS subroutine (e.g. Math.sqrt) is treated as a regular VM function, and can be called by the generated VM code using the regular call command call ClassName.subName nArgs (e.g. call Math.sqrt 1)

OS implementations

Emulated: If you execute / test the generated VM code on the supplied VM emulator (recommended in this project), there is no need to worry about the 8 .vm OS files: The supplied VM emulator features a built-in implementation of all the OS subroutines.

Native: In project 12 we will implement the OS in Jack, and compile it, resulting in the 8 .vm OS files. If you wish to translate a Jack program to assembly, compile the program folder (.jack files) into .vm files, add the 8 .vm OS files to the same program folder, and apply the VM translator to the folder.

Implementation notes #6: OS

Some OS routines come to play during compilation. In particular, the compiler handles...

- Multiplication (*) by calling the OS function Math.multiply()
- Division (/) by calling the OS function Math.divide()
- String constants by calling the OS constructor String.new(length)
- String assignments like x = "cc ... c" by making a sequence of calls to String.appendChar(c)
- Object construction by calling the OS function Memory.alloc(size)

Note: The compiler generates VM code, and the OS routines are implemented as VM functions.

So, every call above is implemented using the regular VM command call OSfunction nArgs