

Foundations to Computer Systems Design
Professor V Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 10.2

The Jack Compiler – Handling Variables

Welcome to module 10.2, we are in the Jack compiler, we are discussing the backend here and in this particular module we will tell you the intricacies of handling the variables which are part of this Jack compilation, so what is it that the compiler has to do when it encounters a variable and as I mentioned to you there are variable sort of four types, it can be the class variables namely field and static and it can also be the, for every subroutine you have variables which can be argument and local variables.

So how are we to handle these variables and that is what we will quickly look at in module 10.2 so though this is specific to Jack, this is the way even in a very high complex programming language like C, C++, C sharp, etc. this is how these variables are going to be handled so this gives you a broad introduction to how compilers in general handle variables and specifically we will be educating you through the example of the Jack programming language.

(Refer Slide Time: 01:36)

Compilation challenges


- Handling variables
- Handling expressions
- Handling flow of control
- Handling objects
- Handling arrays


NPTEL



Module 10.2: The Jack Compiler - Handling Variables PROF. V. KAMAKOTI
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken IIT Madras

Variables



High-level (Jack) code

```

class Foo {
  ...
  field int a, b, c;
  static int x, y;
  ...
  method int bar(int a1; int a2)
  {
    var int v1, v2, v3;
    let c = a2 + (x - v3);
    ...
  }
  ...
}

```

→

VM code (pseudo)

```

...
// let c = a2 + (x - v3)
push a2
push x
push v3
sub
add
pop c
...


```

In order to generate VM code, the compiler must know (among other things):

- Whether each variable is a *field, static, local, or argument*
- Whether each variable is the *first, second, third...* variable of its kind

Module 10: The Jack Compiler - Handling Variables
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

PROF. V. KAMAROTTI
MIT



So we know start with, so I am using the slides from the website, chapter 11, so right so what are the variables, now you see that we have a class named foo and there are three field variables a, b, c, there are two static variables x and y, we explained in the previous module what is the difference between a field variable and a static variable, now there is some method, in this there are two variables a1 and a2 which are basically arguments, there are v1, v2 and v3 which are local variables.

Now look at one expression that let c is equal to a2 plus x minus v3 like this is very interesting c is a field variable, a2 is an argument, x is a static variable, v3 is a local variable so that statement c equal to a2 plus x minus v3 involves all type of variables that are present in the Jack language right, now so when we are compiling these variables should be stored somewhere so that we access it right.

So what will happen here is when foo as a class is created, there will be a location, there will be three locations allocated for foo, the field variables of foo namely a, b and c and in the static memory so if you look at the memory map, location 16 to 255 are to be allocated for static variables, so this x and y will be mapped on to some part of that static so when you are compiling, you know fix that a, b and c are going to be allocated some memory and the starting address of this a, b, c when you are inside class right there is something called the this pointer, that this pointer will point to where the variables of that class are stored.

So when you are compiling the code you ensure that whenever I am executing something related to a class right, something related to an object right, object of a class now that object would have been created, for that object, for all the field variables we will have some locations created and those field variables will be in some consecutive locations right and the starting address of where that object is stored is that starting address is in that this segment right, there is a this that segment that we have seen that this segment actually stores the starting address of where the variables, values of the variables are stored in the memory for the current object.

So keeping that in mind similarly the argument will tell you when I am executing method bar as you see here the argument a1 will be stored, there an segment, argument segment we have already seen a1 will be stored in the zeroth location, a2 will be stored in the first location and there is a local segment in which v1 will be stored in the 0, 1 and 2 now this will point to some memory location say thousand, in 1000 a will be stored 1001 b will be stored 1002 c will be stored.

Now given this type of an info and similarly static without loss of generality let us say 0x is stored 1y stored, now with that let us see what we need to do here so I want to do c equal to a2 plus v minus x3, so now we go into our expression evaluation, in the virtual machine which is a stack based virtual machine I hope you remember the virtual machine that you have done, what do you do, we push a2, push x push v3 sub when I do this sub, x gets subtracted by v3, so x minus v3 is computed and that will be on the top of the stack.

So what will be in this stack, a2 and x minus v3 will be on top of the stack now I say add when we say add, x minus v3 gets added to a2 so you get no and what will be on top of the stack after add, a2 plus x minus v3, now pop c, so a2 plus x minus v3 gets popped down to c, so this is how c equal to a2 plus x minus v3 is basically executed, now this needs to get substituted so when the program is executing where will be a2, a2 will be in the argument stack, argument segment and that location 1.

Argument section location 0 is a1, location 1 is a2 right, so a2 will be pushed argument 1, push a2 is equivalent to push argument 1, push x is equivalent to push static 0 without loss of generality assume that x is stored at 0, pushed static 0, x can be stored anywhere and that index will be known and then push v3, V3 is a local variable so in the local segment v1 is 0, v2 stored

at 1, v3 stored at 2. so push local 2 then sub and add are like this and c, c is a field variable and so where will c be stored, that this segment will point to a address, from that address the second location 0 to address there if this is pointing to say 2500, 2500 a will be stored 2501 b will be stored 2502 c will be stored.

So pop this 2 right so this essential statement, this vm code, this is the actual high-level jack code that gets changed into this vm code sudo level but finally this is the very important, so somehow we have to now translate this c equal to a2 plus x minus v3 onto this code, the last one here and for doing that this mapping of a2 to argument 1, x to static 0, v3 to local 2, c to this 2, this mapping needs to be present right and that is something which is very very important.

So the challenge in handling variables is to actually see how to arrive at this mapping between what is given as a code here and what is finally decided from that, so what we will be discussing in the next few minutes here how are we going to do this mapping, so let us see how are we going to do.

(Refer Slide Time: 08:25)

The slide is titled "Variables" and features the NPTEL logo in the top right corner. It is divided into two main sections. On the left, under the heading "High-level (Jack) code", there is a code block for a `Point` class. The code includes fields for `x` and `y`, a static field `pointCount`, and a method `distance` that calculates the distance between two points. On the right, under the heading "Variable properties:", there is a list of properties: name (identifier), type (int, char, boolean, class name), kind (field, static, local, argument), index (0, 1, 2, ...), and scope (class level, subroutine level). Below this list, an orange callout box contains the text: "Variable properties: • Needed for code generation • Can be recorded and managed using a symbol table". At the bottom of the slide, there is a small video inset of a man speaking. The footer contains the text: "Module 10.2: The Jack Compiler - Handling Variables", "PROF. V. KAMAROTTI", "© 2014", and "Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken".

```
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));
  }
  ...
}
```

Variable properties:

- name (identifier)
- type (int, char, boolean, class name)
- kind (field, static, local, argument)
- index (0, 1, 2, ...)
- scope (class level, subroutine level)

Variable properties:

- Needed for code generation
- Can be recorded and managed using a **symbol table**

Module 10.2: The Jack Compiler - Handling Variables
PROF. V. KAMAROTTI
© 2014
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

This is going to be very straightforward right so when I look at a variable as such I have explained you in the previous module 10.1 itself every variable has a name let us say x, x has a name which is x, then there is a type associated with it, in this case it is int but there are in general in the Jack language we could have int, we could have char, we could have boolean or it

can be another class name also for example you take other the type associated with this is a point right, so it is a class name also.

So int, char, boolean and identifier if you look at the grammar these are all the different type right now kind for example I already told you there are 4 kinds, so field for example x it is kind this field, point count it is kind of static, other, if you look at this other it is kind this argument right and dy if you ask its kind is local, so I have field static local argument and then index, where is it stored in that particular segment, in argument segment, 0, 1, 2 so I need an index and of course there is a scope, all the field variables have within that class and all the subroutine variables has local and argument within that particular subroutine right.

And the static variable of course has across all instantiations of the class so these are the things, and this is at least what we do is we create something called a symbol table where in every variable that you are encountering when you are basically doing this compilation of a class right we actually create a symbol table, we create two symbol tables, one for this class, another for the subroutine right, the symbol table, why we need to do this distinguishing thing is that the symbol table for a method is existing only when I am compiling that method but the symbol table I am doing for the class you exist while I am compiling all the methods.

So I need to have one when I am compiling so every Jack file will have one class and while compiling that class I create one symbol table for the class variables and that will exist till that entire compilation is over and for every subroutine that we are encountering we will create a symbol table for the subroutine for exclusively for each sub routine and that subroutine symbol table will go off as soon as we finish compiling that subroutine.

So every fresh subroutine I encounter whether it is method, whether it is constructor, whether it is function, every subroutine I encounter I basically create a subroutine symbol table for each one of them right, so this is how we go about.

(Refer Slide Time: 11:31)

Symbol tables: construction

High-level (Jack) code

```
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));
  }
  ...
}
```

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

class-level symbol table (field, static)

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

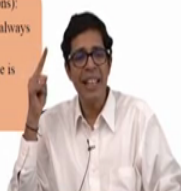
subroutine-level symbol table (argument, local)

In methods only (not in constructors and functions):

- in addition to the explicit arguments, there is always an implicit argument:
- argument 0 is always named this, and its type is always set to the class name
- Generated by the compiler

Module 10.2: The Jack Compiler - Handling Variables
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

PROF. V. KAMAKOTTI
NPTEL



So this is a symbol table, so I see x y point count so what we do here is we are now creating the symbol table so this is the class symbol table that you see on the top and this is the you know the subroutine symbol table for distance alone like and for every subroutine I will create and destroy it once I finish so what is there in the class sub routine, the class sub routine see, note that there are three variables x y point count, all the types are int right, all the types are int as you see here and then the kind the x and y are of kind field while point count is of kind static and then x and y are going to be in 0 and 1 and point count in the this segment, they are going to be in 0 1 in the this segment well point count is going to be in the static segment here we have assumed without loss of generality zero but what will happen in practices when we are compiling an application it will have several class files, so let us say class file one, class file two, class file three, class file four so we will compile in some sequence, it can be in any sequence but when I am compiling classified one that I will encounter some static variable.

So we will allocate say three, there are three static variables so 0 1 2 will be allocated there so when we are comparing the next class file we start from where we left in the previous so that there needs to be some sort of a communication between one class file and another class file in terms of where I am going to store the static variables right so there is a class file called bat, just say this there are two classes called bat and ball right.

So the bat has some three static variables that will be allocated 0 1 2, when I am compiling ball when I find another static variable then I should allocate three for it right so in long sense when you start looking at compiler you will be studying something called linking, linking is nothing but trying to establish some communications between different files which eventually make one executable right, you would have worked on dot C file say there will be several dot C files right which will eventually make one executable, for example even in your Hello World there is some stdio dot H file right you are using printf as a function, you do not know what printf is right, so you just use it and who knows what printf is, the stdio dot H file actually knows printf.

So when you are compiling you take your Hello World code and the compiler also takes the stdio code where the printf routine is there and links it right there is a linking that is happening so what are the different aspects of linking, linking is trying to bring, link several executables together write several files together so this is one very simple example of linking right so that you can keep in mind.

So here without lot of generality we have said that point count is zero but in principle the point count can be three or four depending on how many other class files or a Jack files you have compiled for this particular application before compiling this particular file right, now coming to the next one now we are talking of point when we are compiling this method so there is one argument which is other it is of type point right, it is a type is a class here and the name of that is, name of the variables other and it is at 1.

Similarly dx and dy are local variables, dx dy int they are of kind local and they are in 0 1 now what is this this right already I told you that whenever a method is executing there is a need for accessing the field variables like what we have seen in this case right, there is a need for accessing a field variable in this case I am accessing C, so that means as a method I need to know where that variable C is stored so I need to know where that this is basically stored okay so that is why whenever we are calling a particular method we also pass the reference, that is where that particular object is stored that this will point to where x and y are stored, this is a you know we have already seen this is a particular 16-bit variable now in this this I store 1500 that means x is stored at 1500, where it is stored at? 1501, so but I need to know where this is when I am compelling.

So by default whenever I call a method that this is stored as argument 0 and everything else will be stored from 1 2 3 so here it has only one argument so there are two or three arguments then it will have argument one argument two argument three, so one of the things that we need to understand here is that we need the variable this as a part of your argument because whenever this method is going to access a field variable of that instantiation of the object it needs to know where that field variable is stored, of course static it will know but it needs to know where the field variable is right, so this is very very important. So keep this in mind when we are constructing the symbol tables.

(Refer Slide Time: 17:31)

Symbol tables: usage

High-level (Jack) code

```

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));
  }
  ...
}

```

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

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

↓ look-up

source code example:

```

...
let y = y + dy;
...

```

compiler




VM code

```

push this 1 // y
push local 1 // dy
add
pop this 1 // y

```

Module 10.2: The Jack Compiler - Handling Variables
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken






Now that we have constructed the symbol table right now it is very easy for us to now go and start, it is very easy for us to go and start compiling these classes right so to just sum up one class level symbol table is created with the top one as you see here and then that is used for the entire when start compiling from this to the end of this class here but then there is a symbol, there is another symbol table that is created here which is for every method so when a method is entering that symbol table comes into existence when the method is completed that symbol table is a erased right.

So we will have something called a current symbol table and for subroutine, current subroutine symbol table but we will have one class symbol table which will exist for the entire duration of the compilation of this particular class file or Jack file okay.

(Refer Slide Time: 18:33)

Symbol tables: usage



```
High-level (Jack) code
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));
  }
  ...
}
```

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

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

look-up

source code example:


```
...
let y = y + dy;
...
```

compiler

VM code

```
push this 1 // y
push local 1 // dy
add
pop this 1 // y
```

Module 10.2: The Jack Compiler - Handling Variables
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken



Now every time that you see say let y equal to y plus dy this is a source code in this case, now immediately what we do we push y then push dy and then add and pop it back to y, that is what we do right y equal to y plus dy how do you execute it on the stack based virtual machine, you first to push y, this y, then you then push dy then you add right, push y push dy and then you add then again you pop it to y back right.

Now what is y, y immediately you go to, this symbol table is created before you come here so let us class him so before I start let us say let y equal to dy is somewhere here after this okay now by the time I start executing this the class symbol table will be created and for the current method that symbol table will also be created okay for the current method right so now so this will be available so the moment I see y, push y, where is why I go back it is field one, then I say push dy, dy is local one, add and pop y, y is again this one. So by using this symbol table I can translate this push y, push dy add and pop y 2, push this one push local one add and pop this y, so this is the very simple things right.

(Refer Slide Time: 20:17)

Handling variables's life cycle

Static variables:
Seen by all the class subroutines;
must exist throughout the program's execution

Local variables:
During run-time, each time a subroutine is invoked, it must get a fresh set of local variables; each time a subroutine returns, its local variables must be recycled

Argument variables:
Same as local variables



Field variables:
Unique to object-oriented languages; will be discussed later.

Implementation note
The variable life cycle is managed by the VM implementation during run-time; **the compiler need not worry about it!**

High-level (Jack) code

```
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));
  }
  ...
}
```

Module 10.2: The Jack C... Variables
PROF. V. KAMAKOTI
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken



So this basically we are now talking of the life cycle, the life cycle of a field variable is for the entire class right and so if it is a field variable every object instantiation from the start to the end this scope of this variable is there when you are compiling the class from the start to the end that the life cycle of the field and static variables are there, the case of local variables and arguments it is for within that particular method or constructor or function right so the life cycle is within that right.

(Refer Slide Time: 21:02)

End note: handling nested scoping



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

- Some high-level languages feature unlimited nested variable scoping
- Can be handled using a linked list of symbol tables

symbol tables → ... → scope 2 level symbol table → scope 1 level symbol table → method level symbol table → class level symbol table

Variable lookup: start in the first table in the list; if not found, look up the next table, and so on.

Module 10.2: The Jack C... Variables
PROF. V. KAMAKOTI
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken



Symbol tables: usage



High-level (Jack) code

```
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));
  }
  ...
}
```

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

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



source code example:

```
...
let y = y + dy;
...
```

compiler

VM code

```
push this 1 // y
push local 1 // dy
add
pop this 1 // y
```

Module 10.2: The Jack Compiler - Handling Variables
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken



Now you know in complex programming languages like C right you have, you can declare variables anywhere here we have now restricted to declaring variables at the beginning of the class and here also we are restricted to beginning within the subroutine we have restricted to declaring all the variables at the beginning right but this variable declaration can happen anywhere, it is insane right so I could have then within, so the scope if you remember what you have done in C, the scope of a variable is within the block in which it is described.

So for example if I say for int I equal to 0, I less than something only within that for loop that I will be I declaration is valid, outside it, it will not be valid so but so the moment I have this type of nested scoping right then that basically I will have multi levels of symbol table and then we have to use that, in general in a complex programming language we land up with these type of multi levels of symbol table but in Jack we are not doing it we are no restricting one symbol table for class and one current symbol table for the current subroutine that is being executed right.

(Refer Slide Time: 22:26)

The slide is titled "Compilation challenges" and features a list of five items. The first item, "Handling variables", is preceded by a green checkmark and a small circle icon. The other items are "Handling expressions", "Handling flow of control", "Handling objects", and "Handling arrays". In the bottom right corner, there is a video inset showing a man in a white shirt and glasses. The slide also includes the NPTEL logo in the top right and footer text at the bottom: "Module 10.2: The Jack Compiler - Handling Variables", "PROF. V. KAMAKOTI", "© 2014", and "Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken".

Compilation challenges

- Handling variables
- Handling expressions
- Handling flow of control
- Handling objects
- Handling arrays

Module 10.2: The Jack Compiler - Handling Variables
PROF. V. KAMAKOTI
© 2014
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

So in the will now proceed on to the next module where we will be talking about how to handle expressions, I hope you understood this, I like to see more postings coming on the website, we have seen some people are asking doubts but there are not that many doubts that we anticipated, hope either we are very clear which we are happy at least you post saying whatever you have taught is clear right will be we just want to ensure that you are with us in the course right, we will now go on to module 10.3, thank you.