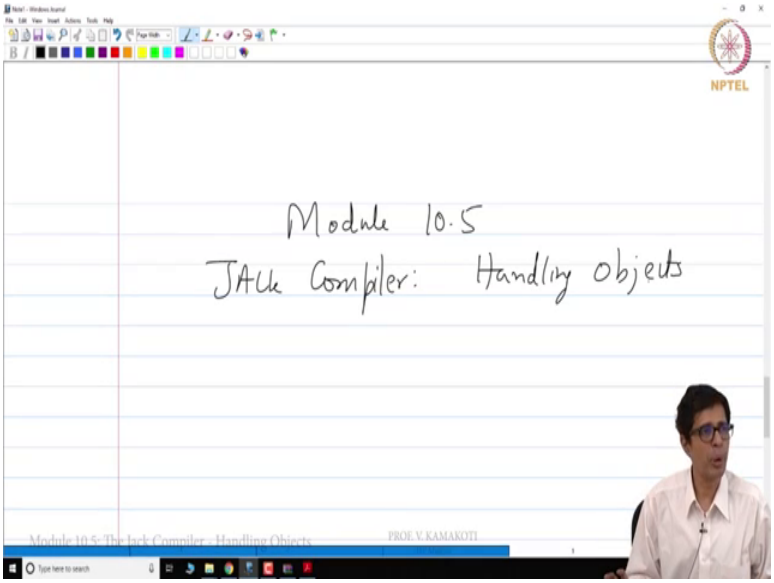
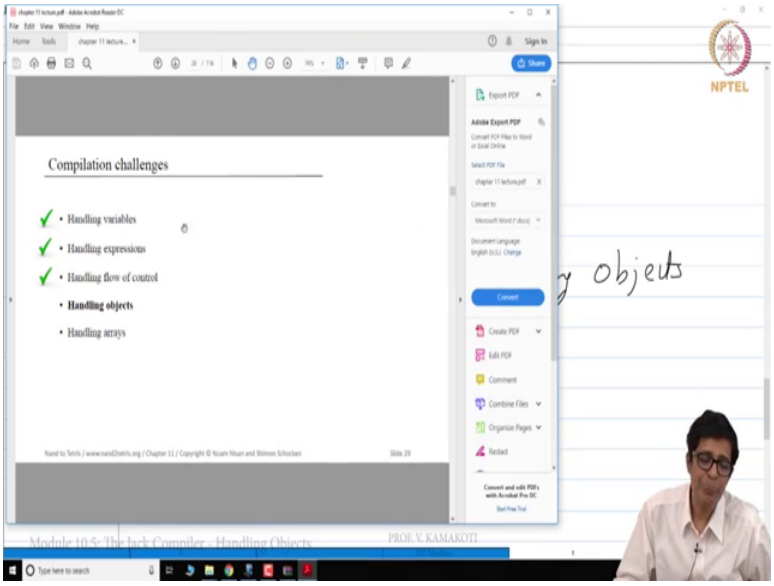
Foundations To Computer Systems Design
Professor V.Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 10.5
The Jack Compiler-Handling Objects

(Refer Slide Time: 00:17)



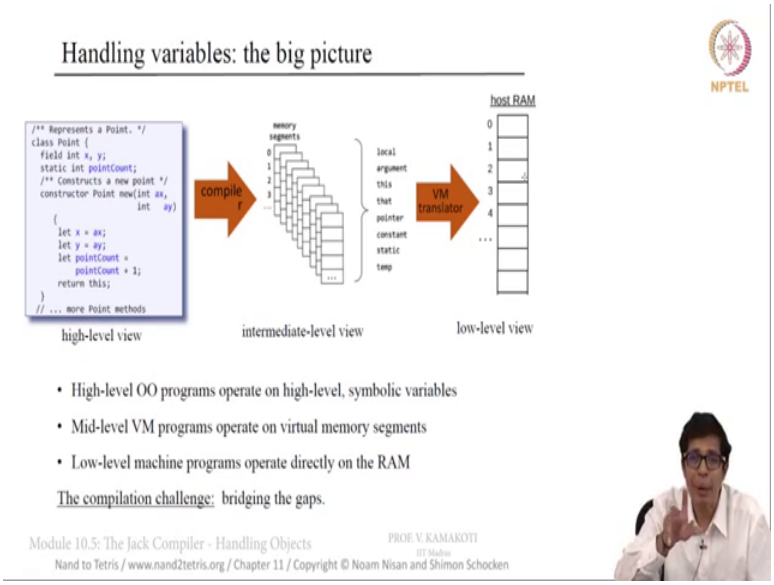
So welcome to module 10.5, in this module 10.5 will be talking about Handling Objects. Now what are Objects?

(Refer Slide Time: 00:26)



Objects are basically instantiations of the class so I have a single class and I could have multiple objects right so now let us see how do we handle such objects right.

(Refer Slide Time: 00:40)



Now the moment we see I have class point, file int x, y static int point count now our construct for that class it is called point new int a x into a y let x is equal to a x, y equal to a y let point count equal to point count plus 1 return this and then go of right so this the Jack word and we will see how this going to get model down to this right.

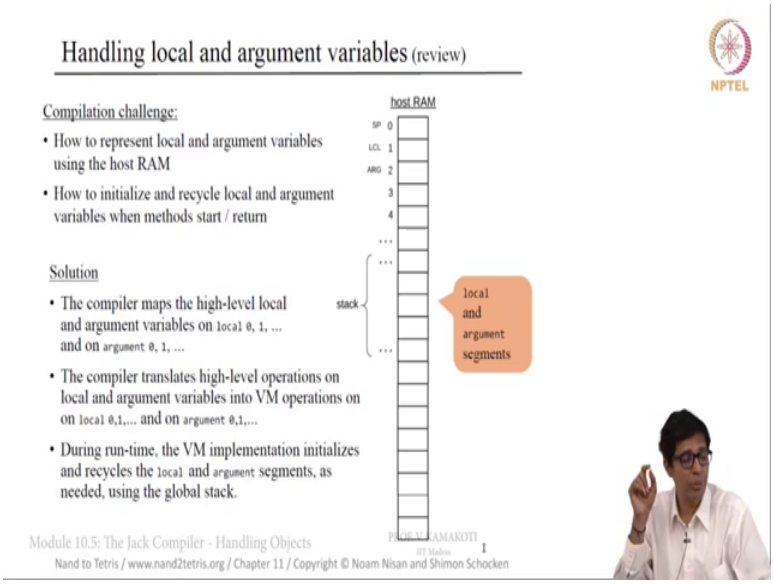
Now we are going to convert it to the VM level program so what are the things? So there are several memory segments that we have already seen there is a local segment where this local variables will be allocated and that local segment is for every subroutine right so when I change from one subroutine to another subroutine the local segment will change. There is an arguments of again that argument is for every subroutine when I go from one subroutine to another subroutine it will change. This is again for every object so when I am executing a subroutine of a particular object every subroutine is associated with an object and so this is for every subroutine of this.

But it is actual used as a temporary for doing certain array computations we will see some good use of that as we proceed on this. When pointer is basically used to adjust the values of the this and that segment right so pointer is zero will go and change what is inside this pointer 1 will change the start address of the that segment so that is and then ofcourse we have constant which

we can go from 0 to 2 power 16 minus 1 static segment is common to all the module so there is only one static segment staring at 16 and going upto 255 and temp segment there are some 8 temp registers which we can use.

So this is all the things that we have seen here so this are all different memory segments and this memory segments are not so finally and the machine vary will one linear memory it will go from 0 to 2 power 16 minus 1 and the segment should be back down to different parts of this memory and ofcourse the start of the segments would be stored in the stack and first few locations of the memory will have the start of the current local, current argument, current this current that etc will be varying the (())(03:11) so we have seen this memory map when we were discussing the VM and also the assembly so now this is the overall so from here to what you see on the leftmost side to the rightmost side this is how the entire variables of an object need to be handled. So we use all this segments to actually map on the variables of the objects that you instantiate.

(Refer Slide Time: 03:42)



Now let us see how these things are gone. See on the stack always that we have seen the local and argument, local and argument variables are always on the stack and why it is necessary because when I move from one local to another local right one (argu) when in move from one subroutine to another subroutine the local and argument of one subroutine should be now stored and the next local and argument has to come so we have seen this when we are doing the VM and so the local and argument segments are always on the stack and the current local current

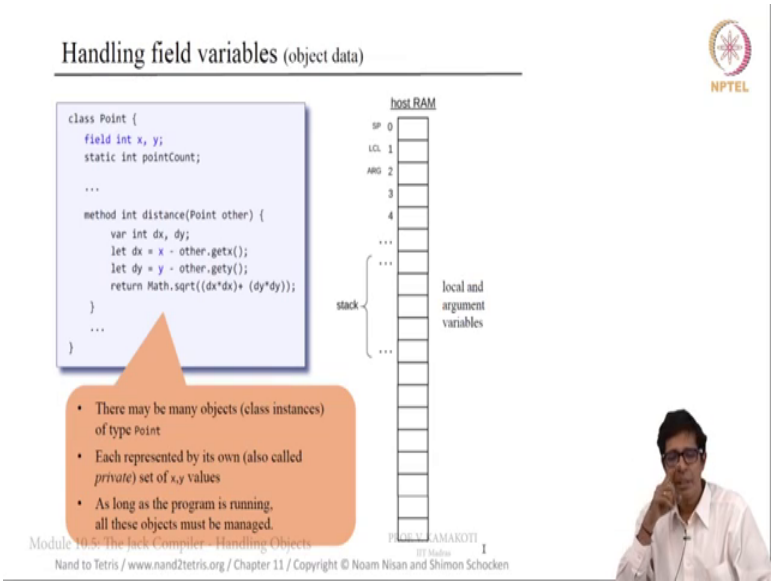
argument entries are there in location 1 and 2 of the RAM, this we have seen in the earlier past itself right.

So when I, when so this is automatically taken care by your VM so when I do a call there is a calling function, the calling function (have) has its local and argument segments and there is a called function which will have the local for so the call at the VM the call VM actually takes care of so if you just say call the implementation of call at the VM level will take care of setting up the local and argument segment of the called function and when the called function completes and comes back right it responds the return will take care that the, the local and argument segment of the calling function is taken care of.

So function A calls B, function A has its local and argument on the stack when it calls B that call it will of the VM right so there is a VM instruction call if you just invoke that call the way we have implemented call will take care of taking this entire thing and the way the call is implemented in the VM will take care of setting up the local and argument segments for the called subroutine that is B. So when B starts executing its local and argument will be there ok and the way so when B return there is return statement in VM and the return statement will take care of reviving back the local and argument segment of the calling function.

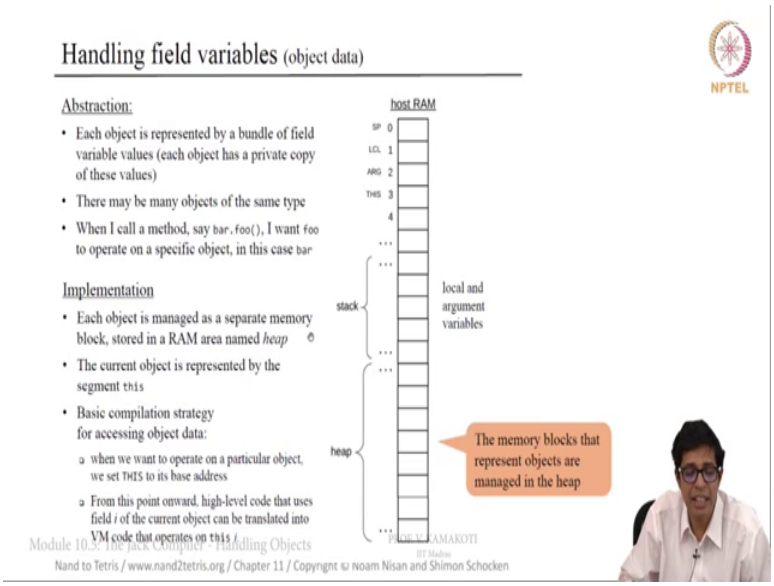
So the movement of the local and argument segment from the called function to the calling function and calling function to the called function the adjustment of this local and argument segments is automatically taken care by the way we have implemented call and return in the underlying virtual machine right that we have taken care so if you go back to module 7, 6,7,8 there we had actually done this type of an implementation and I hope you remember those if you do not please go and revise that particular part ok.

(Refer Slide Time: 06:53)



Right, and as I mentioned earlier the field variables, the filed variables are going to be part of the this.

(Refer Slide Time: 07:07)

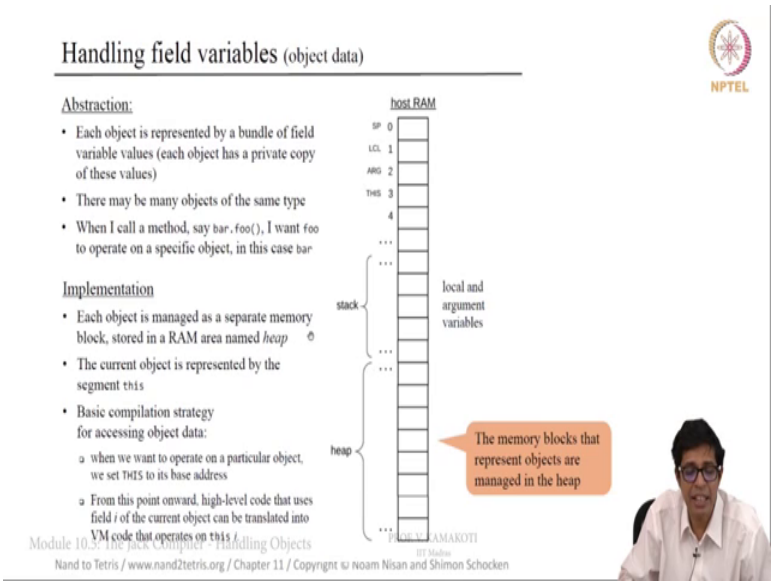


So every time the location 3 in the RAM holds the pointer to the current object which is being handle if I executing a subroutine of some object K the object the start address of where that object is stored is there in this ok. Now what will happen is that this actually stores the base address of that object so that is how the field variables will be mapped onto this 0, this 1, this 2

etc so in the case of point x is this, 0 and y is stored in the this segment with 1. So use that this to basically handle the field variables right.

Right, that is what we are saying here when we want to operate on a particular object we set this to its base address from this point onward high level code that uses field I of the current object can be translated into VM code that operates on this side so if I want to access filed variable 1 I have to access this 1 so we have already seen that and this is how the field variables arrange.

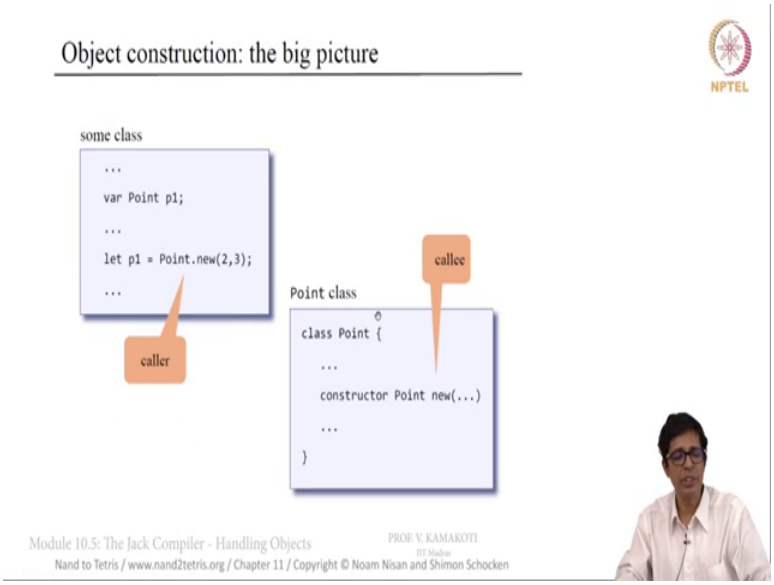
(Refer Slide Time: 08:19)



So the other thing is that if I want to access RAM right if I want to access keyboard or whatever that we have already seen push 8000 pop pointer 0 so the now that this will now, this will now have 8000 and now I say PUSH 17 yes then POP this , 1 so in 8000 one 17 will come.

So look at this code PUSH 8000 right POP pointer 0 so 8000 goes and sits in this now if I say PUSH 17, 17 gets pushed on to the stack right then I say POP this 1 so this is 8000, 8000 plus 1 right this is an index in that segment so 8001, 17 will go and sit so this is how I can access any memory routine I want ok. So now when we are looking at handling objects there is something like constructing a object and also manipulating the object let us see both of this in great detail right.

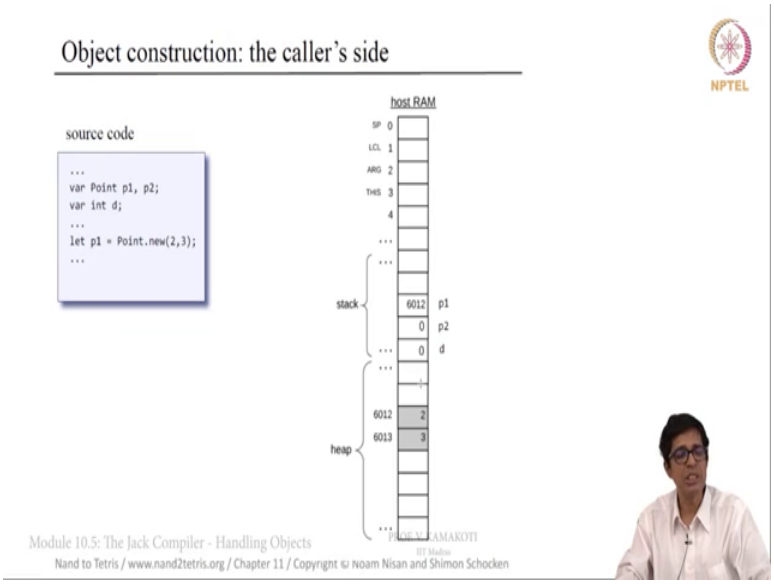
(Refer Slide Time: 09:34)



Now I have var point P1 right so and let P1 equal to point dot new 2, 3 so point is some class some class in some class I am so ok point is a class in some other class I am instantiating a variable of type of type point P1 and I am saying let P1 equal to point dot new.

Now what is point? Point is a class inside that there is a constructor function called point new, so what is happening is there is a caller function which is calling this calling function here right, s how is this object now constructed, so this is how it is constructed.

(Refer Slide Time: 10:21)



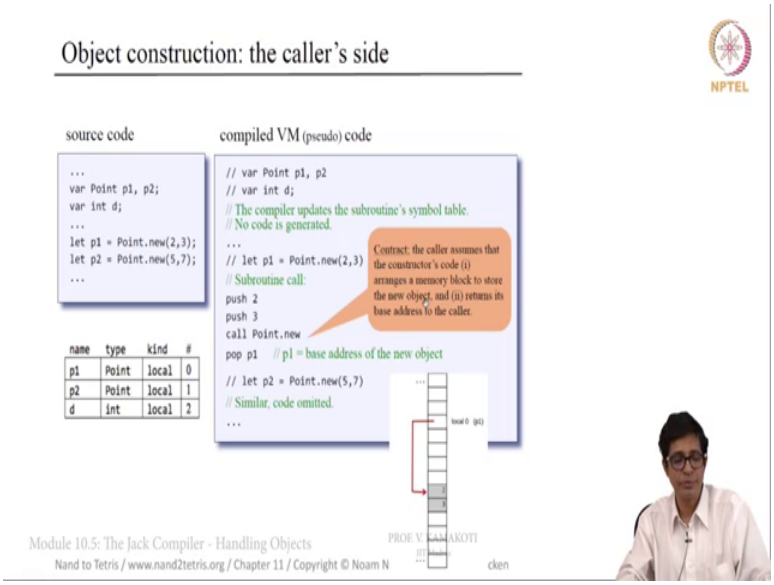
First and foremost we know that this is a constructor function the moment this is a constructor function so let us say this is a stack when this function starts executing this will be in the local variables because this is var so P1, P2 and all are in var this also in the var since P1 P2, P1 is P2 is said to be constructed so it is 0 this is also in 0.

Now what we say is that we say P1 equal to point dot new 2, 3 right now somewhere in the heap so there is a stack there is some heap, heap is the extra memory that is available for you know allocation for, for new objects etc so the operating system will now identify that your P1 point P1 is stored at 6012 and 6013 right. So now P1 if I say what is P1? In C if I say array suppose I say int I of 10 what is I? I is the starting address of that array similarly in this what is P1? P1 is the starting address of where that point is stored. So what will be stored in the stack for P1? It will be stored 6012 now in 6012 when I go and see I will find 2 and 6013 I will see 3.

So essentially this means you are constructor function what it is supposed to do? It is now it needs to take some memory from your heap allocate two entries in that heap and pass on the address of the first entry and that will be stored in P1 so whenever I want P1 dot x I go to P1 I get the first address then dot x I know that will be field variable 1 so I will go to P1 and then dot x will be field variable 0 so P1 plus 0 6012 I will get 2 there and similarly for y I will get 3 here.

So the essentially what we need to do here is the moment I say point dot new 2, 3 now two locations are allocated in the heat on which I go and store 2 and 3 in addition the first locations two consecutive locations are allocated in the heap in which we go and do 2 and 3 and the address of the first location is stored in back in P1. So this is what we mean by constructing an object.

(Refer Slide Time: 13:16)

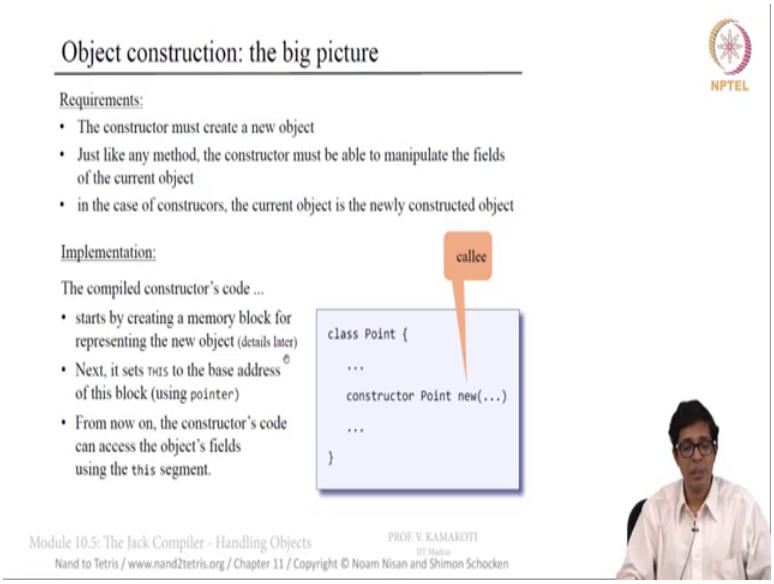


So what essentially happens here right so what it means so when I say let P1 equal to point dot new of 2, 3 this is the VM code that you will be getting here first I will push 2, I will push 3 then I will say call point dot new of 2 right so call point dot new of 2 the essentially that means there are 2 arguments that are need to be pushed here right. Now so the contract is the caller assumes that the constructor code arranges a memory block to store the new object and returns its base address to the caller right. So call point dot new will return a base address to the pointer so that I so now so what will happen? Call point dot new, 2 and after this call finishes what will be available the base address of the pointer will be available of the point will be available that is whatever you say and that I say pop back to P1.

So P1 will store the base address where the two points are store right that is what we need here and similarly there is a code here so this is basically what we need to get here ok. Now let us see right so let us now see how this resulting impact will happen so will say var point P1, P2 and d so P1 P2 will be stored as local variables and d also as a local variable and initially everything would be zero when we say point dot new 2, 3 when we call this function point dot new 2,3 the compiler will that the operating system will allocate 6012 for it and in 6012 and 6013 it will store 2 and 3 and it will return back the value 6012 here right and so this P1 will get the value 6012 then we do point dot new 5,7 again point dot new will go it will go and ask the operating for two spaces the operating system will give 9543 and 9544 for example so it will go and store 5 and 7 and it will return back 9543 in this.

So in terms of implementation what happens is point dot new will return 9543 which will essentially gets stored into P2 here so this is how two objects are basically created. So what we have seen now is what is happening at the caller routine so call nothing I just pushed up parameters I call this function and whatever result I just pope it onto this P1 right that is what we are doing but now we need to do what is going to happen at the callee routine?

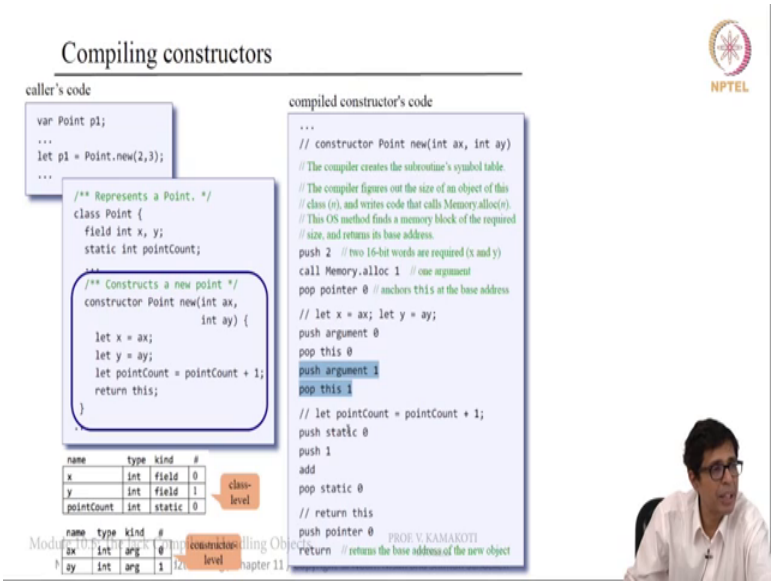
(Refer Slide Time: 16:26)



So what is the callee supposed to do? It needs this is a constructor so it should create a new object and just like any method the constructor must be able to manipulate the fields of the current object right so I create a object (())(16:40) one has 2, another has 3 etc.

Now the current objects and it returns back that object right so it will return back any constructor will return back a pointer to the object it has created right. Then what do you mean by creating an object? Creating space for storing those variables there right, so this is the thing and so it uses that this as a base address and it will return back so let us see how this is going to happen right.

(Refer Slide Time: 17:13)



So this is what is happening so now when point got complied what had happened your x and y are field variables and without loss of generality your point count is stored at 0 location in static, it can be in some other location also.

Now what will now when I say point dot new, 2, 3 this will starts executing so your VM code is something like this right now we will say so first and foremost since it is a constructor there are two arguments to this so we just first say PUSH 2 so this is point dot new right this is function point dot new and what we need to do is we will push 2 in general what will the compiler figures out that you know how many local variables are there in the stack and it will push that many amount of that value here so if I had 10 variables I will say PUSH 10 then I call a function called call memory dot alloc with one argument and what is that argument? 2 so this like memory dot alloc 2 this 1 stands for how many arguments, suppose I have 10 arguments 10 local sorry 10 field variables here then I will say PUSH 10 and I will say call memory dot alloc 1that means it will allocate 10 locations for those 10 field variables and then I say pop pointers 0.

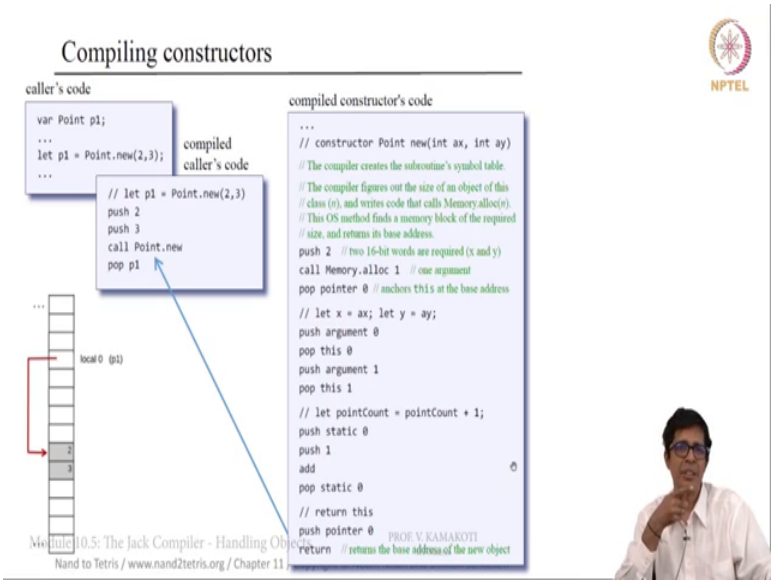
What is pop pointer 0? Pop pointer 0 is that this will become the this address ok now what we do now I say let x is equal to a x so x is equal to a x means push argument 0 so a x is the 0^{th} argument so push argument 0 onto the stack and pop this 0 because x would be in the this segment x is a field variable x will be in that segment and that location 0. So now your x will get

the value of a. Now push argument 1, argument 1is a y and push argument 1 and pop this 1 this will be in this in that this segment the 1 will be the first variable will be y.

So in that this segment 0^{th} variable is x 1 (first) variable is y so or variable number 1 is y, variable number 0 is x so pop this 0 push argument 1 pop this 1 so argument 0 is a x, argument 1 a y so this push argument 0, pop this 0 will do x is equal to a x push argument 1 pop this 1 will do y equal to a y then after that we have to increment point count so let point count equal to point count plus 1 right that is what we wanted to do so push static 0 what is static 0? Static 0 will have the current value of point count right and push 1 ofcourse push constant 1 add so this is whatever is there on the top of the stack gets incremented by 1.

Pop static 0, so the incremented value is stored back into pointCount so this is what, then what we did, what we do we push return this right, so we just put push pointer 0 and then we return so what will be on the top of the stack? The address of the newly created point is will be there and that is what goes into this ok.

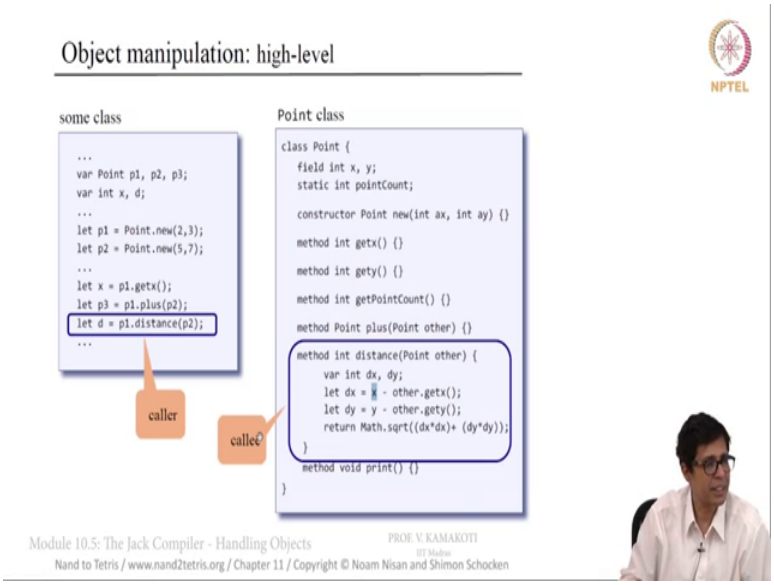
(Refer Slide Time: 21:08)



So this return will go back here and there we say call point dot new this return will go and that I say pop P1 so P1 will get the value of the newly created object. So this is how an object is created and taken back and this is the mapping of that syntax onto the VM.

Now we have understood creation of object creation from a caller and a callee perspective now let us go and so we know construction now let us now talk about manipulation.

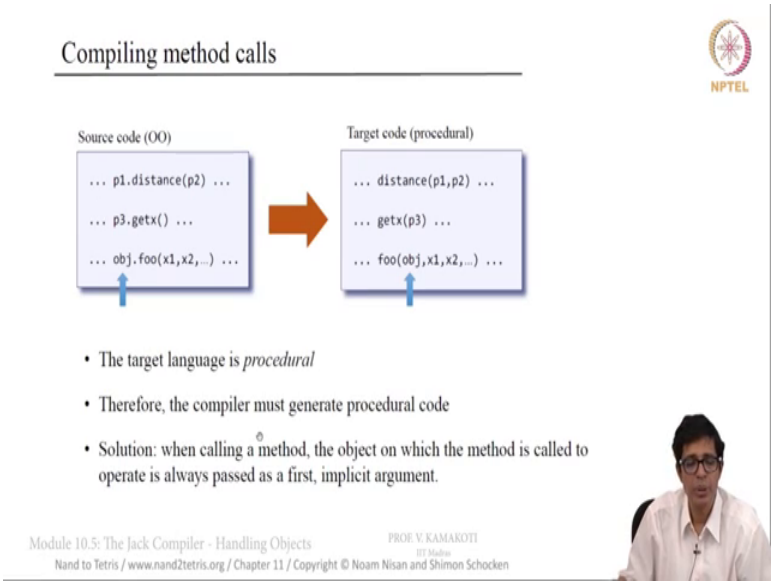
(Refer Slide Time: 21:48)



Now this is here some class I say let d equal to P1 dot distance of P2 where P1 is some class P1 is class point, P1 is a point d is a integer let d equal to P1 dot distance with P2, P2 is another point. Now the distance routine is actually a method int distance point other so other is some other point so what we do is that we find the difference in x coordinate so what is the distance between x, y and x1, y1 and x2, y2? This is x2 minus x1 the whole squared plus y2 minus y and the whole square, square root of that right.

So that is what we are doing so other is another point and then finding the distance of the current point that this P1 from the point P2 that is what this means P1 dot distance of P2 right. So P1 is point P2 is another point I am finding the distance between P1 and P2. So for that I will call the method inside P1 right with the other as P2 so what so d x is a local variable which will be x minus the current point minus the other point x other dot get x and d y equal to y minus other dot get y and what we do is we return the square root of that is (x) d x squared plus d y squared as you see and this is and then we do this so now this is the caller and this is the callee and what will the VM code look like is what we are going to see now.

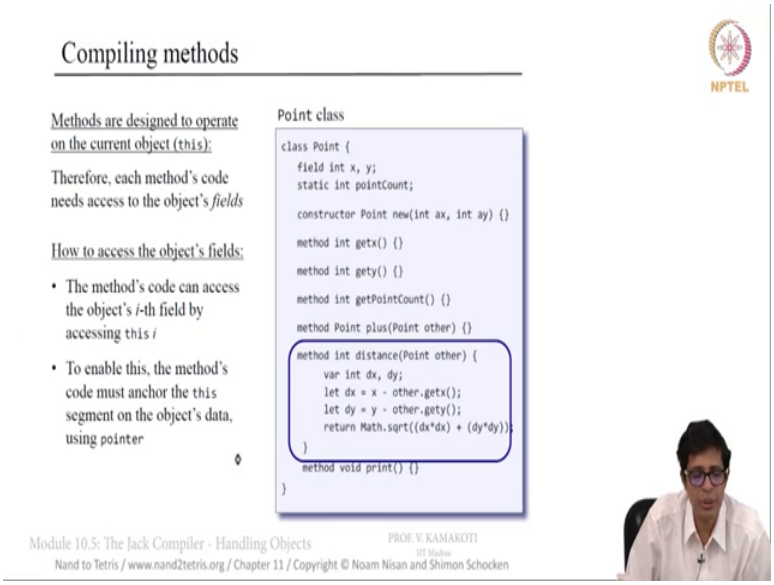
(Refer Slide Time: 23:40)



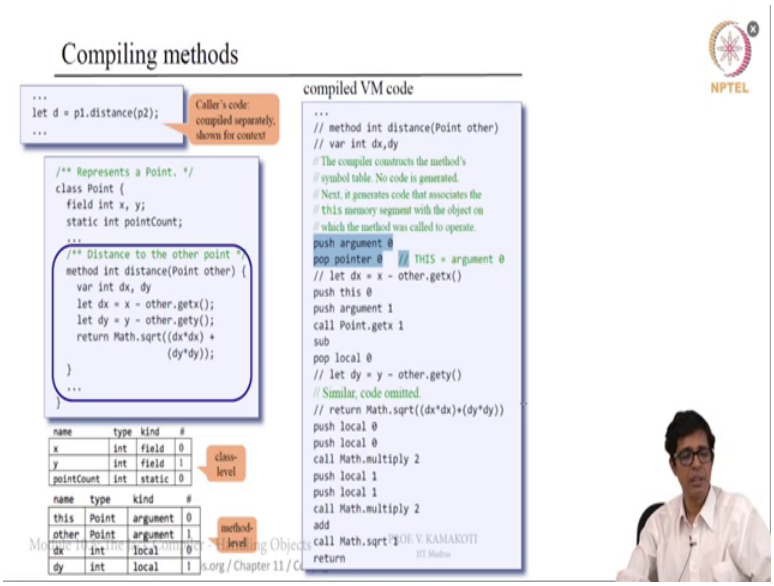
Right, so in general we are calling object dot foo of x1, x2, x3 some object and within that object there is a routine foo of x1 x2 and actually in the target code there is foo of object x1 x2 ok so this is how this is going to happen right. So when I call object dot foo of x1 x2 what we need to do is we push the object first because we need and we push x1 we push x2 all this things because we need to know which object is necessary so we push object we push x1 we push x2 etc ok for example let d equal to P1 dot distance of P2 we push P1 right and then we push P2 also which is the argument then we say call distance call point dot distance or whatever and that will return a value again you pop it to d right.

Ok so how do we go about doing this? so this is the caller and this is the callee now what is going to happen?

(Refer Slide Time: 25:01)



(Refer Slide Time: 25:13)



Ok now right let us look at the callee thing right, so again this is point dot distance so what I do is first and foremost I push argument 0 because I may want to use the field variables inside right so if it is a method I may want to use the field variables of this point which I am using in this case right when we say when I call P1 dot distance I am using x and y which are field variables of the current object P1 right, P1 of instance point right and so this is an object of type point and I am using the field variables associated with P1 so I need the this right so what I am doing here as I told you I am pushing argument.

So whenever I am calling this that is what whenever I am calling object dot foo I first push object and then push x1, x2 etc then call foo. So in this case I first push P1 and then I then push P2 then I call this distance. Essentially that means distance with this thing so essentially the argument 0 that you see and the argument 1 you see right argument 0 is P1, argument 1 is P2 right, so that is what we see here first I say push argument 0 because argument 0 would have come from there and that will point to where P1 is right and then I pop pointer 0 right that means now I am handling P1 when I say d equal to P1 dot distance of P2 the distance routine corresponds to P1 right.

What is now we are trying to do is that d equal to P1 dot distance of P2 now when this is calling when this P1 dot distance of P2 the way it is compiled is first we will push P1 then push P2 and then we will call distance that is what we have seen here push P1, push P2 and will call distance so what will be argument 0 here? The argument 0 will be pointer to P1 argument 0 will be 1 will be pointer to P2 right. So before this point dot distance is going to execute what we will see is that on the stack we will see P1 as argument 0 pointer 1 as argument 0 and pointer P2 as argument 1 right.

Now what happens is we call that distance routine so we start executing this part of the code now, now this is that part of the code now what we are going to see here is first and foremost this particular routine corresponds to which object P1 right so when I am executing this routine and when I say x this x corresponds to P1 x so that means that whenever I am accessing any field variable inside this distance it should correspond to P1 so that is why please note here that the first step would be push argument 0 so what is argument 0? It is a pointer to P1 and that argument so the pointer P1 is pushed onto the stack and that is pop backed to 0 that means the current this whatever you are seeing as this in the current this will now point to P1.

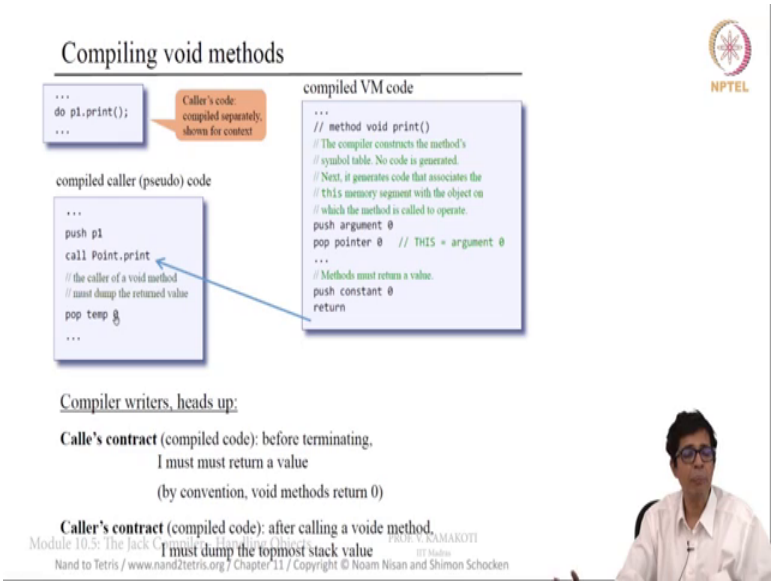
So if I am going to access any field variable then it corresponds to P1 x and P1 y so that is the two statements. So whenever I am calling a method we need to ensure that the calling routine the caller pushes the whenever I am calling a method corresponding to a particular variable it pushes that variable which will be the pointer to the start of those field variables and then it also pushes ofcourse the other argument, the 0^{th} argument will always be a pointer to that object for which we are going to on which we are going to execute whose method we are going to execute. Please

note that method is corresponds to every object every instantiation has its own associated method ok.

So P1 dot distance of and P2 dot distance are different in the sense that when I am doing P1 dot distance this x corresponds to P1 x and y corresponds to P1 y while P2 dot distance if I do then this x will correspond to (P1) P2 x and y will correspond to P2 y, so after doing that all the other things are same so what we need to understand is when we do this two statements are very-very important if these two are done only the corresponding field variables will get from the corresponding object right otherwise we do not know from where it will get. So this is one thing that after that it is all quite easy like so point pop pointer 0 this will be argument 0 so now I will say push this 0 call argument 1 call point dot get x 1.

So push this 0 means what? So we are getting get x right get x of other dot get x now I am pushing this 0 push this 0 means what? I am pushing x here push argument 1 is what? Argument 1 is other right P2, P2 is argument 1 so push argument 1 then call point dot get x right 1 so this argument 1 is so point dot get x of P2 so let us again do this push this 0 will push x into the stack push argument 1 will actually push P2 into the stack because argument 1 is P2 argument 0 is this thing then call point dot get x right will then which basically call point dot get x with P2 so that will return this call will basically return the x value of P2 right and then we do sub, so it will do x minus the P2 x then pop local 0, local 0 because d x is this thing so pop local 0. Similarly d y equal to y minus other dot get y will do the same thing so then we will say push local 0 call math dot multiply 2 so it will do local 0 is d x local 1 d y so this will do d x squared will be on the top of the stack d y squared will be on the top of the stack will add both of them so and then call dot the addition will now after this add d x squared plus d y squared will be on the top of the stack now I will do square root I will get this.

(Refer Slide Time: 33:38)



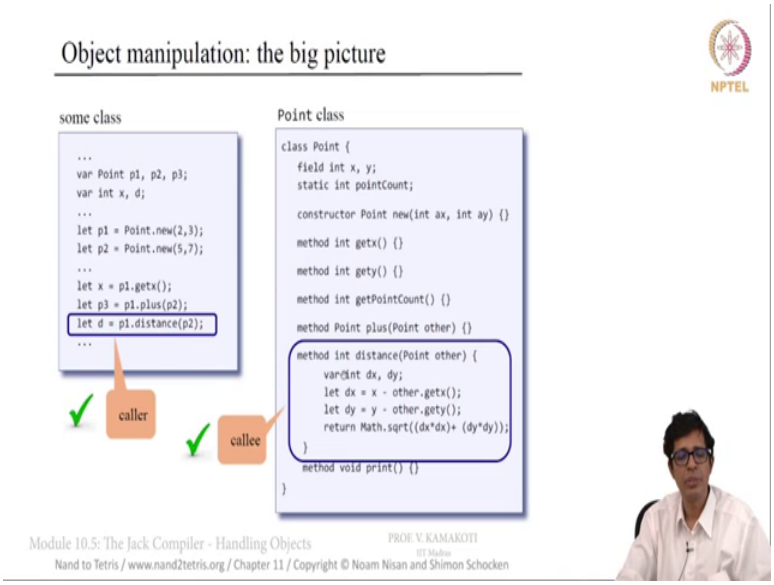
Ok so this is how this routine gets compiled so most important thing that we need to see is the first two statements namely push argument 0 and pop pointer right and the return value goes back and so call point dot distance the return value goes back and then that I pop it onto d so d gets the distance between these two points right. So but this are methods which have a return value but there are methods which do not have a return value which normally we do using do. So do statement works with methods which do not return anything while lets statements will work with methods which will return something.

So what in the case of do again when I say P1 dot print we push so this will say push P1 and then say it will say call print 1 ok so it will push P1 and call print with 1 argument so again push argument 0 pop pointer 0 so we are now setting up the context of P1 here because we have to do P1 dot print so I am setting up the context of P1 here and then do whatever you want then always we assume that any calling the VM code assumes that any subroutine when it returns it will return some value right so it has the VM routine he way we have implemented the relation between calling function and called function is that the called function always will return something that is how we have implemented VM.

So in the case of do certainly this routine is not going to give anything so we put push constant 0 and then return so this is it is a wide function is not returning anything still I will push constant 0 and return and that will come back. So this is how we if it is wide this is how we compile that

function and this goes back the return goes back here so after we do the return since we are doing do we just call point dot print here and then this pop temp 0 right this is pop temp 0 because this is we are going to use anything so this is returning something I have to pop it out because the stack has some redundant value I need to pop it out, so this is the way we compile void methods.

(Refer Slide Time: 36:09)



So this is how the object manipulation big let d equal to P1 dot distance P2 we call this, this is the caller function we understood how the caller function work and understood how the callee function works right. So this is about handling objects, now in the next module we will see how we are going to handle arrays, thank you.