Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

Module No. # 05

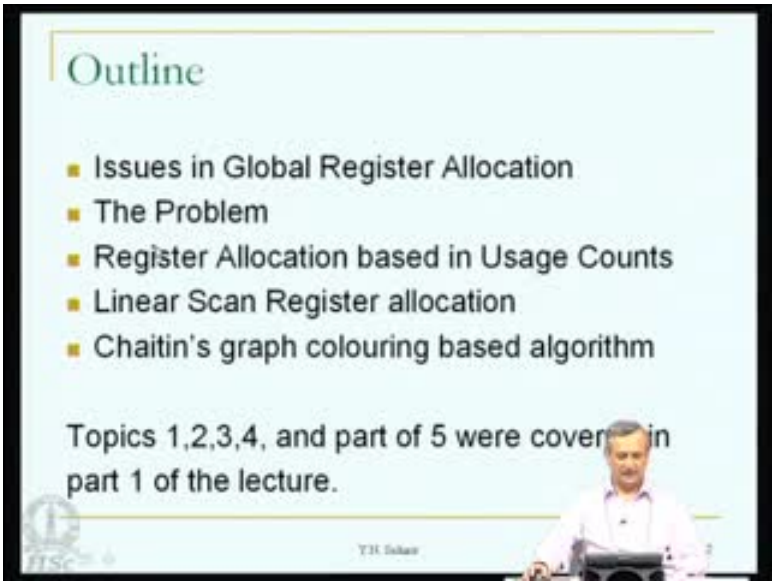
Lecture No. # 14

Global Register Allocation - Part 3 and

Implementing Object-Oriented Languages

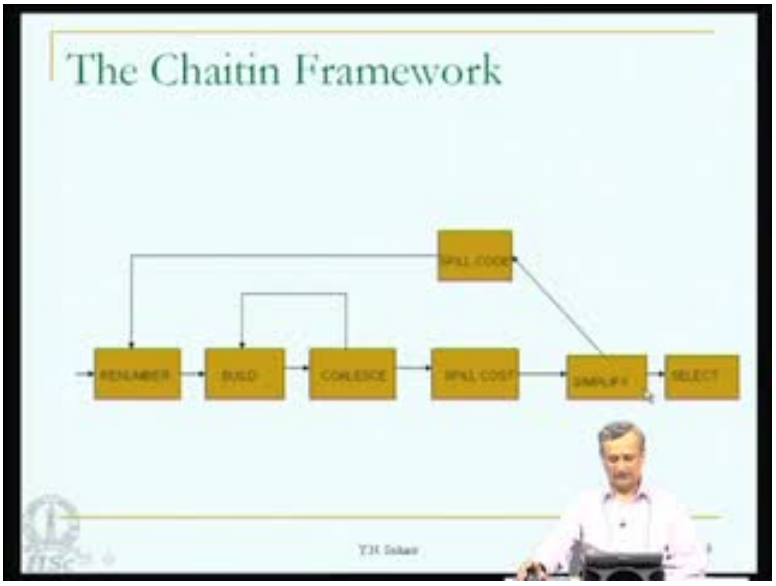
Welcome to part III of the lecture on Global Register Allocation.

(Refer Slide Time: 00:20)



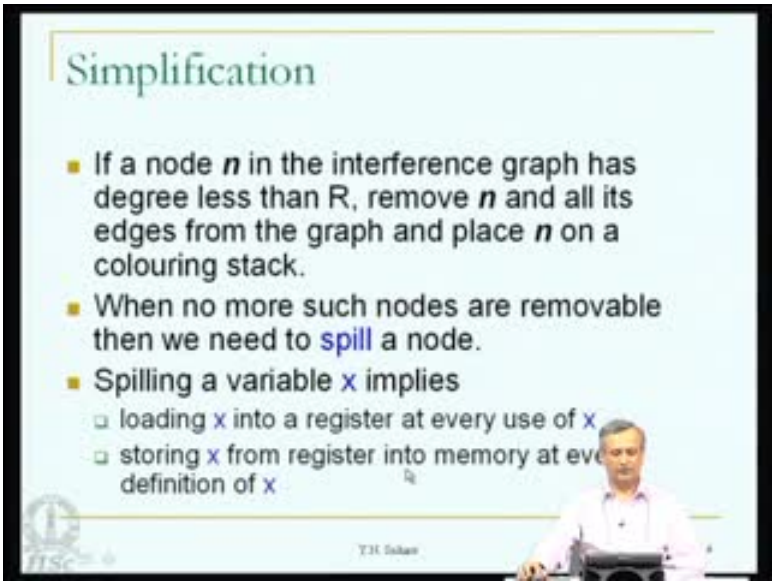
Last time we discussed issues in Global Register Allocation, the Problem itself, Usage Counts based Register Allocation, Linear Scan based Register Allocation and part of Chaitin's graph colouring based allocation.

(Refer Slide Time: 00:40)



To do a quick recap: here is the framework of Chaitin's register allocation algorithm. To begin with, there is a renumbering phase in which the live ranges or webs are identified. Then, there is a build phase, where the interference graph is built. Then, there is copies absumption or coalescing, which removes some of the copies and reduces the size of the interference graph. Then, we have a spill cost computation; the spill cost is computed based on many heuristics. Then, there is a simplification, which removes the nodes and edges and reduces the graph. Finally, this entire thing – for the spill nodes, we have to introduce spill code. The whole process is repeated and finally the selection of registers is made.

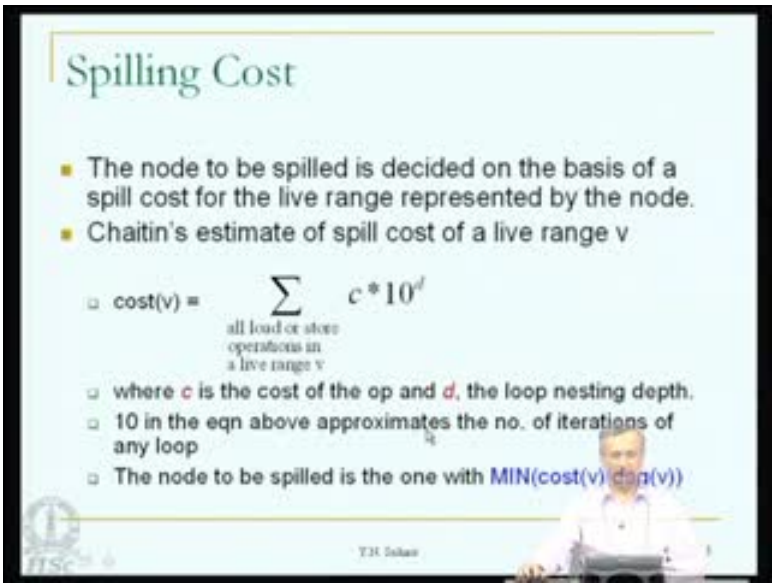
(Refer Slide Time: 01:46)



Quickly, simplification is – take a node whose degree is less than R and then remove the node and its edges from the graph. Keep doing this until it is not possible to do anything more.

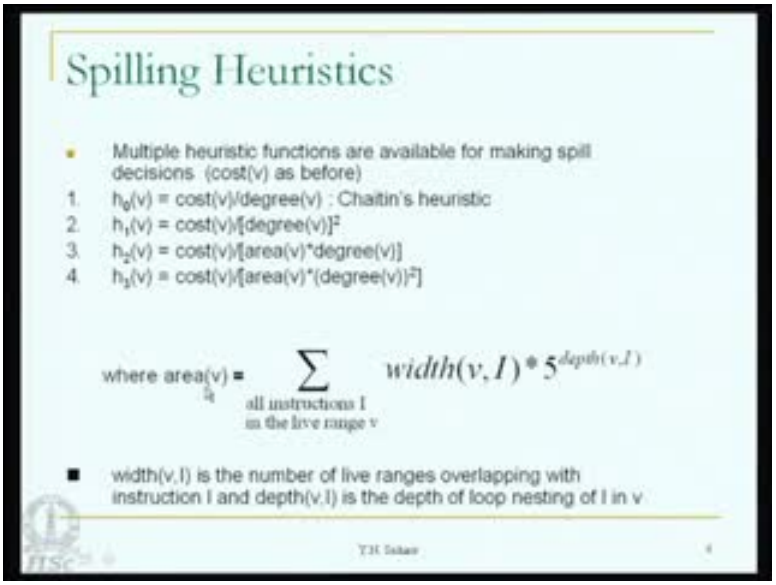
At some point, there will be nodes, all of which are half degree greater than or equal to R – the number of registers available. In such a case, we need to spill a node. Spilling a node implies loading x into a register at every use and then storing x from register into memory at every definition.

(Refer Slide Time: 02:22)



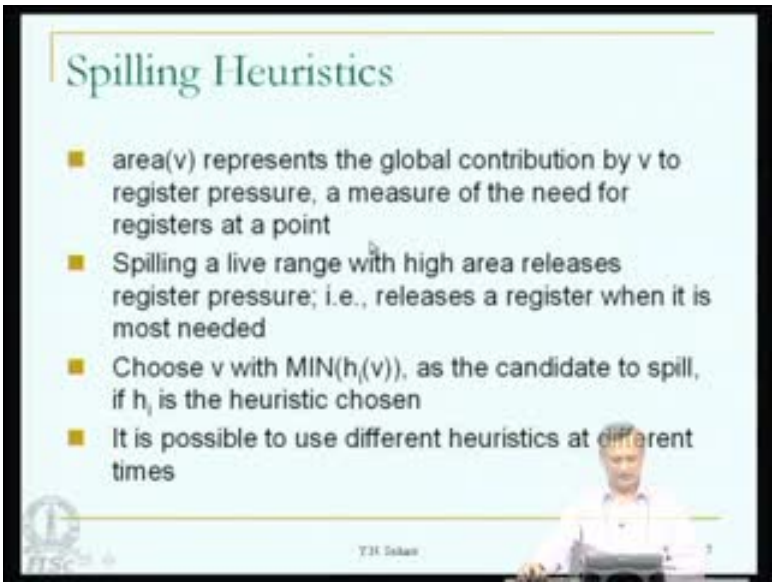
Spilling cost is very simply computed as cost v equal to sigma c star 10 to the power d, where the sigma is over all the loads and stores in the live range, c is the cost of a load or store, and d is the depth of nesting. Basically, this is used to approximate the number of iterations in any loop.

(Refer Slide Time: 02:43)



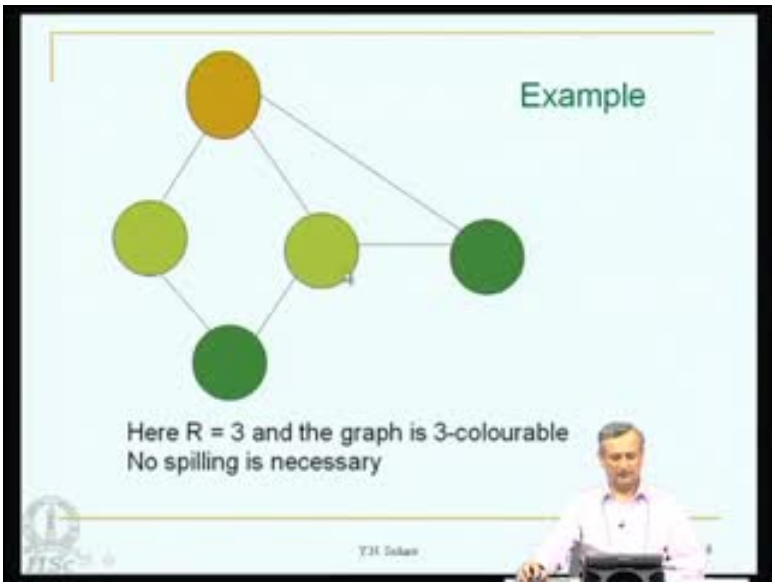
There are many spilling heuristics.

(Refer Slide Time: 02:50)



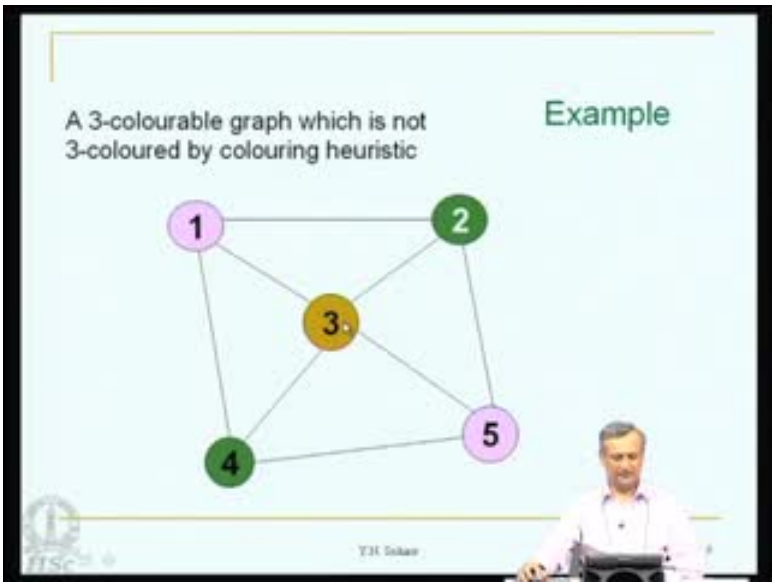
Some of them use this area, which actually represents the global contribution by the live range to the register pressure. It is possible to use many heuristics at different points of time.

(Refer Slide Time: 02:59)



Here is a very simple example, which shows how graph, which is 3-colourable can be coloured without spilling.

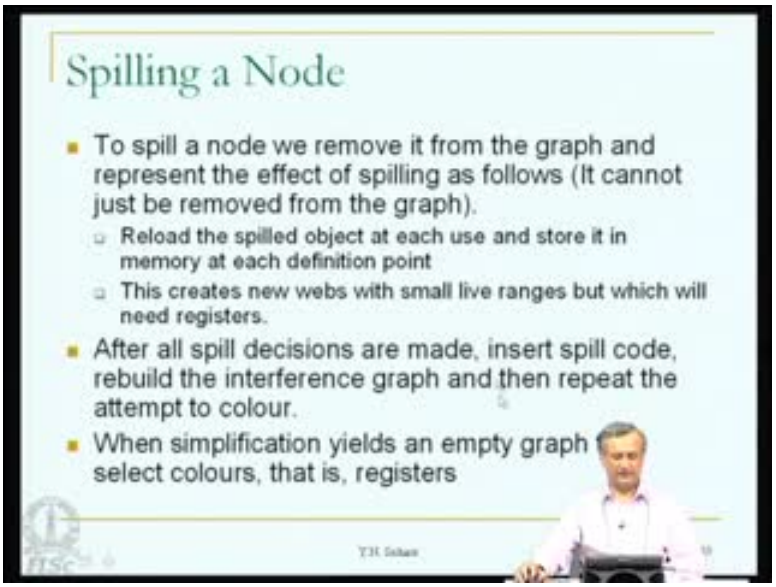
(Refer Slide Time: 03:08)



Now, here is a graph, which is 3-colourable, but it cannot be coloured by the colouring heuristic. It says we will need a spill, but obviously, as we have shown here, it does not

need any spill. So, there are some heuristic improvements possible to the heuristic itself, which will help us in colouring such a graph; we will see that a little later.

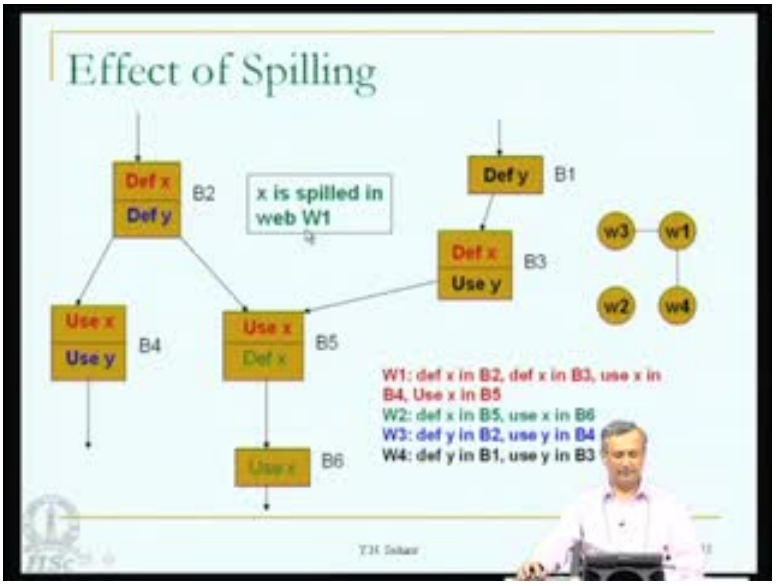
(Refer Slide Time: 03:30)



How to spill a node? To spill a node, we remove it from the graph and represent the effect of spilling by reloading the spilled object at each use and storing it in memory location at each definition point.

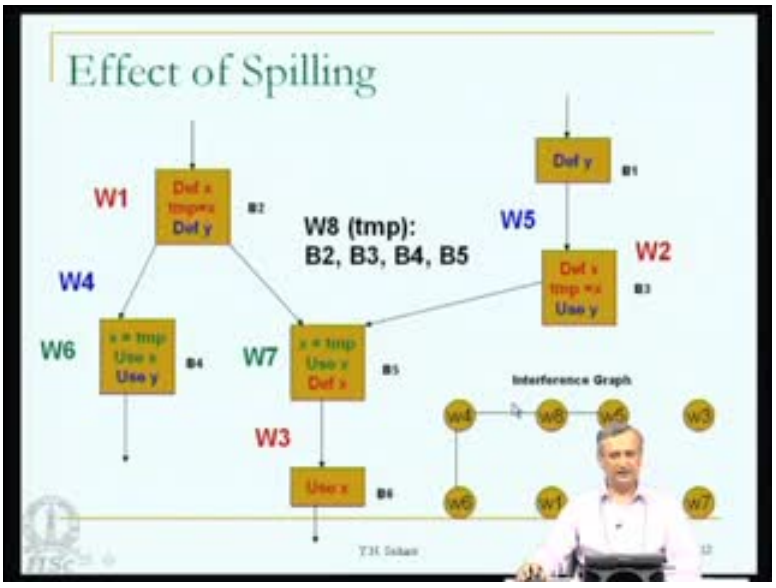
Now, the problem is this creates new webs with small live ranges, but which still need registers. After all the spill decisions are made, insert spill code, rebuild the interference graph, and then repeat the attempt to colour. When simplification yields an empty graph, we stop.

(Refer Slide Time: 04:03)



Let us see the effect of spilling on webs. Here is the web example that we had seen before. There are many webs here: W1, W2, W3, W4 – in different colours. Most importantly, W1 consists of this Def, this Use, this Use and this Def; similarly, the others.

(Refer Slide Time: 04:37)



Suppose x is spilled in the web W1, what happens? Here is effect of that… Before that, this was the interference graph (Refer Slide Time: 04:33) created for this particular web without any spilling.

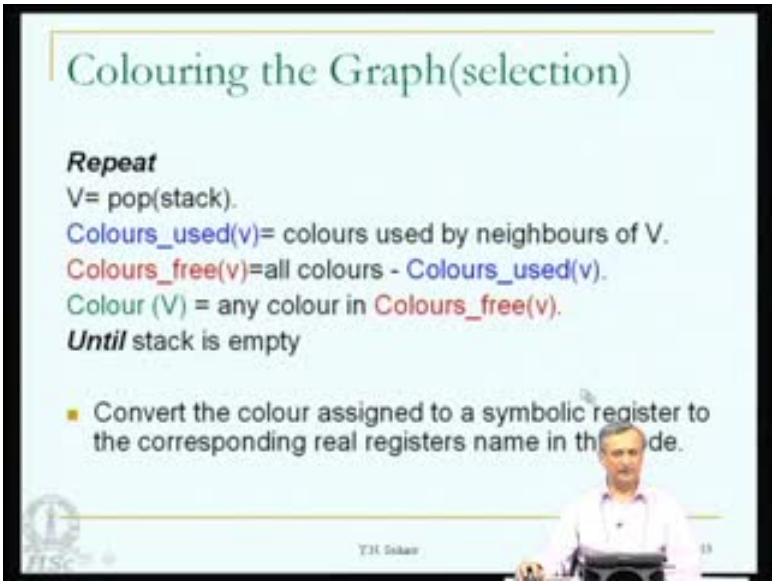
If x is spilled and when there is a Def x, obviously we need to store that x into a location; let us call it as temp. This is a definition point of x. So, we have stored it into memory. Here is a Use point of x and we are actually reading it from memory, x equal to temp and then Use x.

Again, here is another one (Refer Slide Time: 05:07): x equal to temp and Use x. This Def x is again a definition point. So, we are storing it into memory by saying temp equal to x. The others are not spilled. Only this particular web is spilled. So, all the loads and stores are appropriately taken care of.

Now, there are many webs here (Refer Slide Time: 05:28). The webs actually get split into many. Whatever is shown in red; Def x and temp equal to x is a small web on its own. This is x equal to and this is temp equal to x. So, this is a computation and then a Use. So, this becomes a small web on its own. Here is another small web (Refer Slide Time: 05:52), the third small web, fourth small web, and rest of the other webs remain as they are.

Now, there are 8 webs. Then, of course, the temp equal to x, x equal to temp, temp equal to x, x equal to temp – these 4 now become a separate web consisting of these (Refer Slide Time: 06:12) – B2, B3, B4 and B5. With these 8 webs, now, the interference graph slightly changes. In this case, there are many nodes, if you have the same number of edges really speaking. However, in a very large set of webs, in general, the interferences will reduce and the interference graph becomes a little more sparse possibly easier to colour.

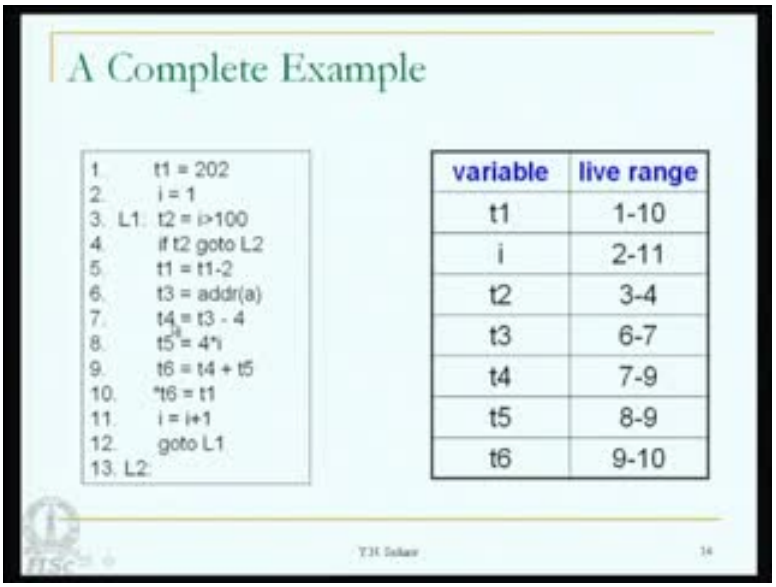
(Refer Slide Time: 06:40)



The colouring heuristic is simple. There is a repeat until loop. All the vertices are now on the stack. So, take a vertex from the stack. Colours used v is colours used by neighbours of v in the graph. Colours free v is all the colours minus colours used. Colour v is any colour in colours free. We keep doing this until the stack is empty. So, that would give us a colouring, which is a fairly straightforward process.

Convert the colour assigned to a symbolic register to the corresponding real registers name in the code. So, we rewrite the code and that completes the colouring heuristic.

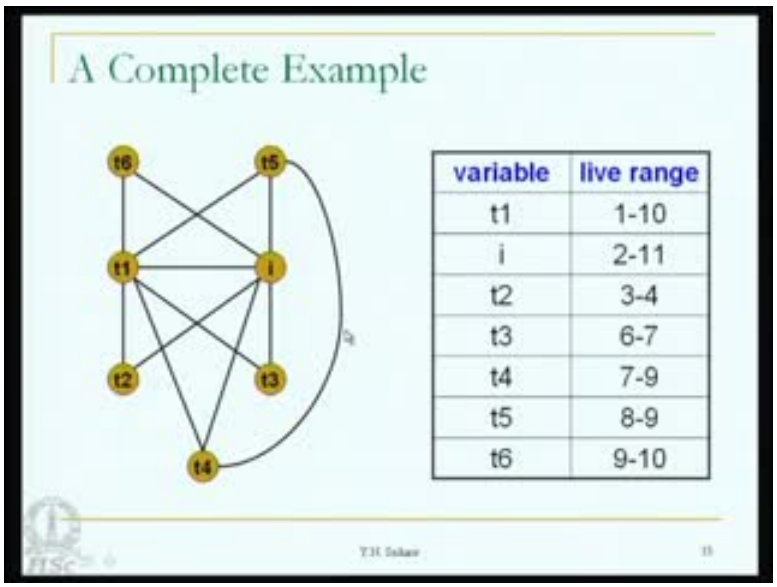
(Refer Slide Time: 07:22)



Now, let us take a complete example of a program – start from identification of the live ranges, building the interference graph, then colouring the graph and so on. Here is a very simple example; the program is the intermediate code. Here is t1 equal to 202, i equal to 1, t2 equal to i greater than 0. Then, there is a test – if t2 go to L2; if we cross 100, then we go to L2; otherwise, we keep looping in this particular loop; t1 equal to t1 minus 2, t3 equal to address a, t4 equal to t3 minus 4, t5 equal to 4 star i. t6 equal to t4 plus t5, star t6 equal to t1, i equal to i plus 1 and goto L1. So, this is the loop.

There are 1 2 3 4 5 6 7 variables here (Refer Slide Time: 08:21) and here are the live ranges of the variables. For t1, the range is from 1 to 10. So, here is a definition, the first definition and the last use is in this number 10. For i, it is from 2 to 11. So, here is the first definition and the last use. So, that is 11 (Refer Slide Time: 08:39). Then, similarly, we have t2, which is a very short live range. t2 is defined here and t2 is used here. So, 3 4; that is it. t3 is 6 7. Similarly, defined here and used here. t4 is 7 9. So, t4 is defined here and used in 9. t5 is 8 9. So, t5 is defined here and used here (Refer Slide Time: 09:01) and that is all. t6 is 9 10. So, t6 is defined here. Star t6 means you read t6 and then do indirection. So, this is really the usage of t6. So, this is the live range of this particular variable, t6.

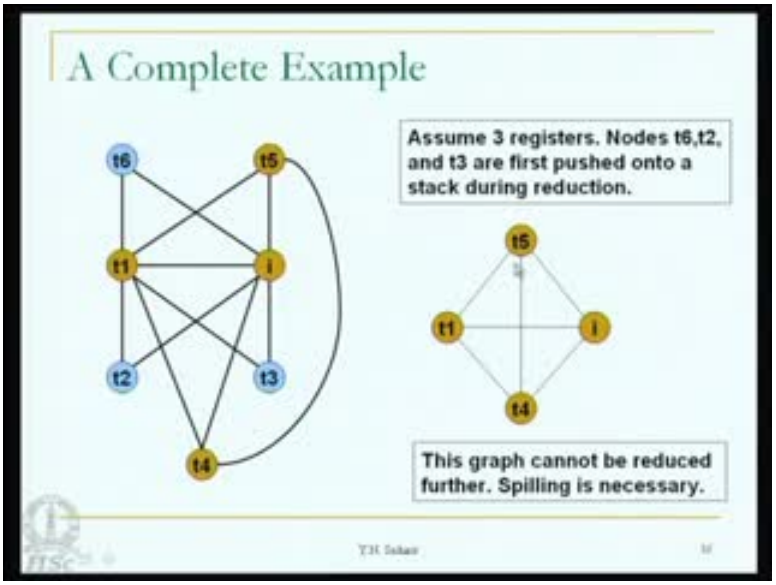
(Refer Slide Time: 09:17)



For these live ranges, it is easy to see the interference. For example, almost every variable over lapse with the live range of t1 and i because they run almost throughout the program – 1 to 10 and 2 to 11, all others are in between; we can easily see that.

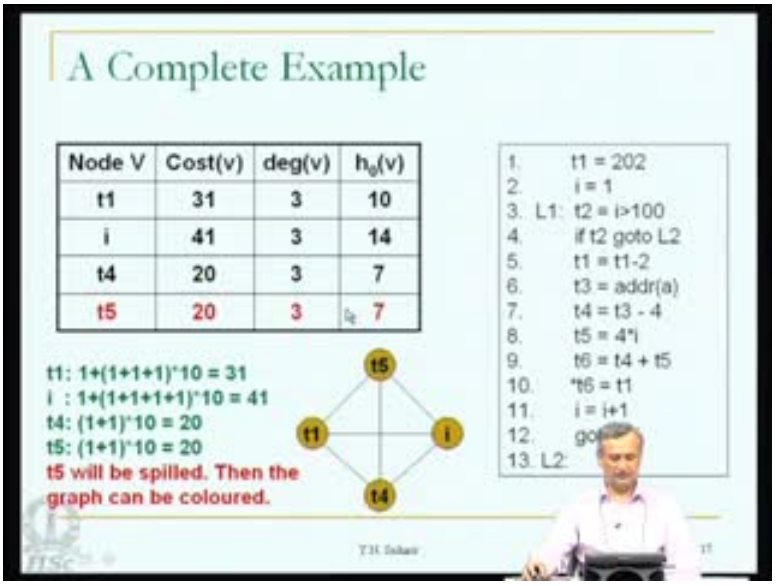
For the two nodes: t1 and i, there are edges coming from every other nodes. Say for example, from t2, there are two edges to t1 and i; from t5, there are two edges to t1 and i, and so on and so forth. Then, similarly, t4 and t5 intersect because this is 7 to 9 and this is 8 to 9. So, there is an overlap. Then, the others for example, t6 does not interfere with any other node except t1 and i, and so on. So, these live ranges are all small, but still colouring this with 3 registers is not a trivial job.

(Refer Slide Time: 10:18)



Let us assume that there are 3 registers and nodes t6… Also, what we do is – pick nodes with less than 3 degree; that is, 2 degree. t6, t2 and t3 have 2 degree. So, these three are actually deleted from this particular graph and pushed on to the stack. That is the first thing. So, they have been shown in blue. After this is over and the edges corresponding to these are all removed, the graph reduces to this. So, this is a complete graph (Refer Slide Time: 10:54) and this cannot be reduced further, but spilling is necessary because every node here has 3 degree.

(Refer Slide Time: 11:00)



Now, we need to compute the cost of spilling using the spilling heuristic. Let us use the simple Chaitin's heuristic of sigma c star 10 to the power d. So, node t1 – let us look at the number of occurrences of t1. These are the only 4 nodes for which we need to compute cost. We need not compute cost for those nodes, which have already been deleted.

Let us assume that the load cost and the store cost are the same. So, t1 occurs once in this instruction number 1, then it occurs in instruction number 5; this is twice really, and then in instruction number 10 as well. So, one is here and then inside the loop. So, we have 1 2 and 3. So, 3 occurrences of t1. So, that is 31; 1 plus 1 plus 1 plus 1 star 10; that is, 10 to the power 1. So, this becomes 31.

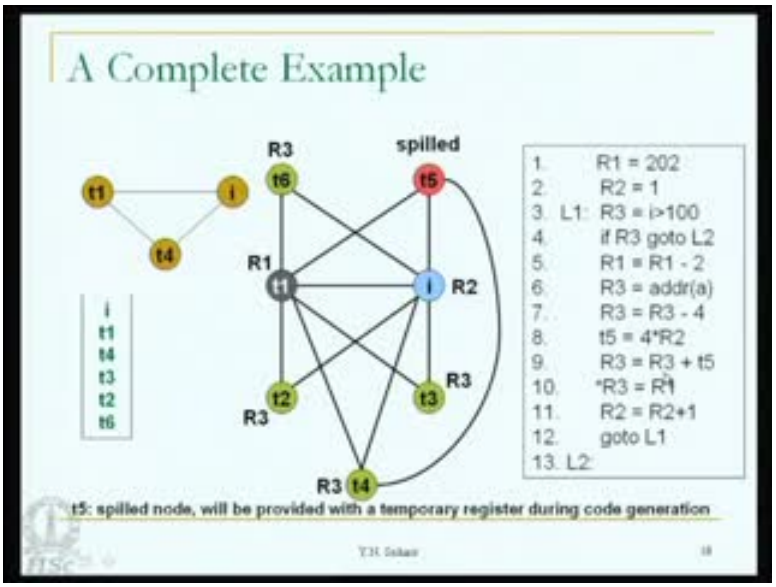
Similarly, for i, here is 1 and within the loop, there is one occurrence here (Refer Slide Time: 12:18), another occurrence here, third occurrence here, and fourth occurrence here. So, there are 4 occurrences. So, there are 4 1's here. So, 1 plus 10 is 41.

t4 has a definition here (Refer Slide Time: 12:30) and a use here. So, there are only two of these. So, 1 plus 1 star 10; that is 20. Then, similarly, t5 is defined here and used here. So, 1 plus 1 star 10; that is 20.

So, the costs are 31, 41, 20 and 20. Out of these two, (Refer Slide Time: 12:53) we could have chosen… and degree is 3 for all of them. Now, h naught v, which is cost per degree

is 10, 14, 7 and 7. We could have chosen any one of these (Refer Slide Time: 13:04) for spilling.

(Refer Slide Time: 13:09)

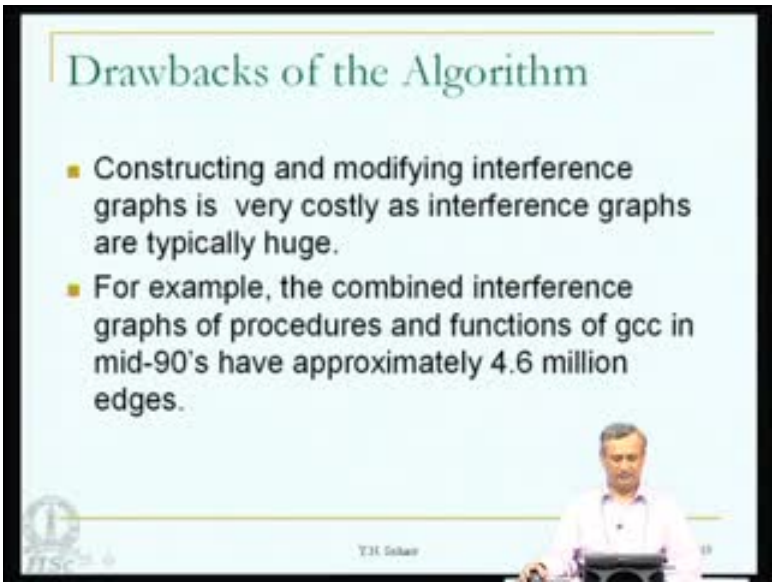


Let us assume that we have chosen t5 for spilling. Once it is spilled, it is always permanently in memory. So, this is the reduced graph. This can be easily coloured with the 3 colours; there is no problem at all. This is the stack of t6 t2 t3. Then, we delete t4, t1, and then i. So, we can easily colour this graph by assigning i a colour, then, we take t1; it is assigned a different colour; that is here, and then we take t4 and we assign a colour, which is different from each of these, and so on and so forth.

Observe that t6, t2, t4 and t3 are all given the same colour; that is, R3. If we rewrite the code using these registers… (Refer Slide Time: 13:58) I have not written assembly code here, but I have just rewritten the intermediate code using these registers so that it is easy to understand. Observe - wherever we have t1. we would have put R1 and wherever we have i, we would have put R2. So, see that this code is correct (Refer Slide Time: 14:15) Even though this 6 and 7, there is R3 equal to address a and then R3 equal to R3 minus 4. So, this is a read of R3 and then write in to the same one. So, this is reused here (Refer Slide Time: 14:28). Because the live range of this R3 and live range of this R3 are not the same… This R3 for example, live range is 6 7 and live range of this is 7 9. So, there is no overlap as such. This is a use and this is a definition. After this usage, we can actually write into the same register. Therefore, this is possible.

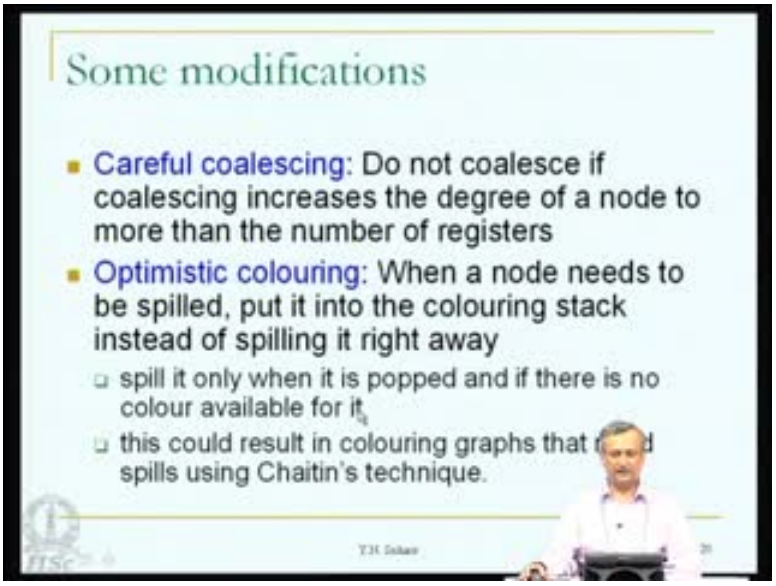
This is the rewritten code (Refer Slide Time: 14:56), which shows that registers have replaced the temporaries, but please observe that t5 is still t5. That is because, it is actually in memory. When we actually generate machine code, this spill node will be provided with a temporary register. So, this will (Refer Slide Time: 15:14) become a computation into a temporary register. Afterwards, t5 will be used here and then it is discarded because it is not needed any more. So, if memory can be used as an operand in the instruction, this does not become necessary. That is the example.

(Refer Slide Time: 15:33)



Now, what are the drawbacks of Chaitin's algorithm? Constructing and modifying interference graphs is a very expensive process as interference graphs are typically very huge. If you take for example, the GCC compiler itself and then for example, all the functions, procedures, etcetera are combined and you try to draw a combined interference graph, then the graph has approximately 4.6 million edges; extremely huge graph. So, this is one of the major drawbacks.

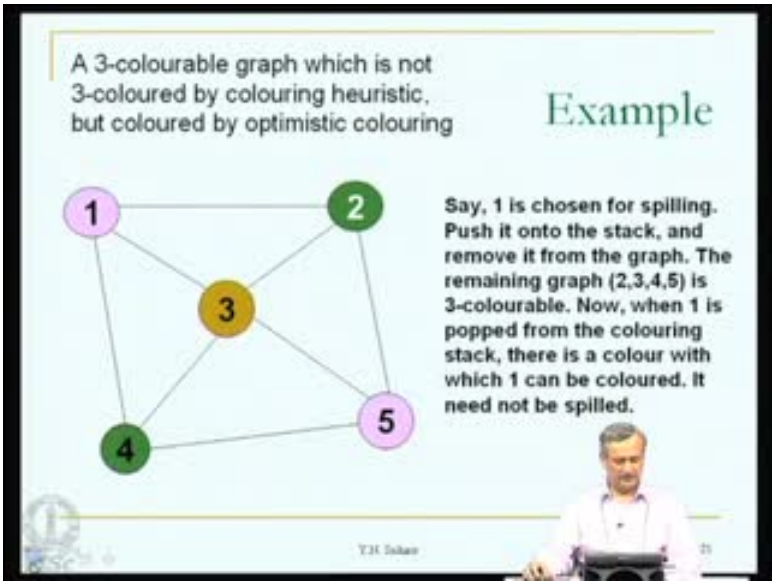
(Refer Slide Time: 16:09)



What are the modifications necessary in order to improve this particular algorithm? You could be more careful during coalescing. For example, do not coalesce if the coalescing increases the degree of a node to more than the number of registers. So, if you do this, then it may lead to a spill. That is why, there is no need to coalesce. This is copies absumption problem.

Optimistic colouring: When a node needs to be spilled, put it into the colouring stack instead of spilling it right away. Assume that it can be coloured, put into the stack, hope for the best. Spill it only when it is popped and if there is no colour available for it. We will demonstrate this very soon. This could result in colouring graphs that need spills using Chaitin's technique.

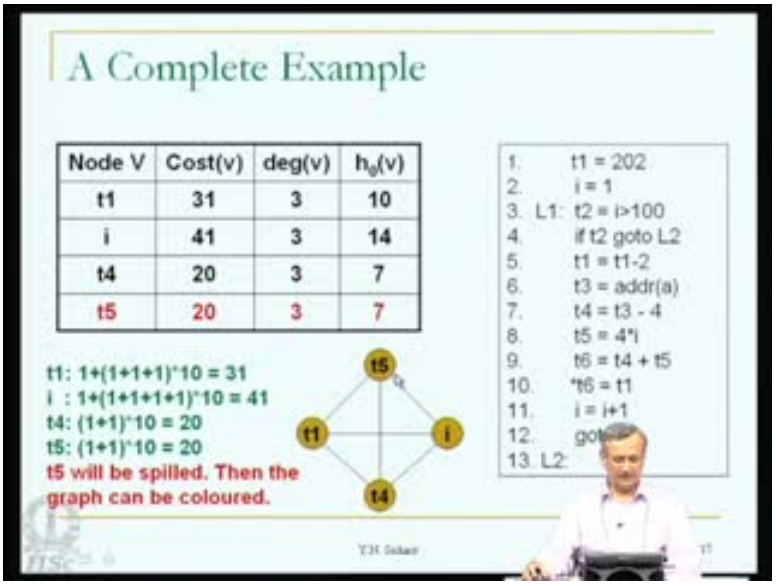
(Refer Slide Time: 17:02)



For example here: This is a 3-colourable graph, which we saw before. It cannot be coloured using Chaitin's heuristic, but using optimistic colouring it can be. Say, 1 is chosen for spilling; one of them has to be spilled; otherwise, there is no way you can colour it with three colours. Do not actually replace this and say – yes it is spilled and replace the loads and stores by appropriate instructions from memory. Push it into stack; assume that it is coloured and push it into stack.

Now, we have the rest of the graph here – 2 3 4 5. These just go away. This can be coloured with three colours. Let us say the colours assigned to the particular graph – 2 3 4 5. 1 is left on the stack. Observe that 2 and 4 have the same colour, 3 has a different colour, there is a colour free, which can be assigned to 1. Because we did not spill 1, but pushed it on stack, colouring of this (Refer Slide Time: 18:14) has been possible. This is the optimistic colouring that we indicated few minutes ago.

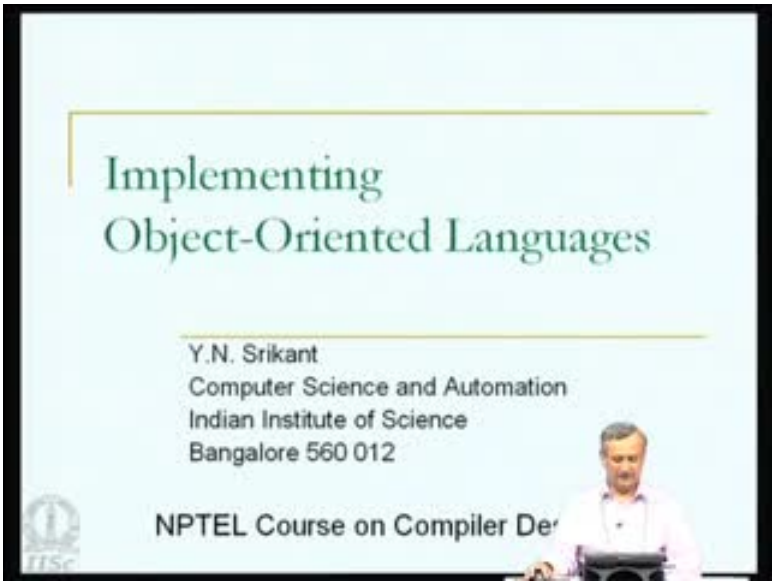
(Refer Slide Time: 18:34)



However, there is no guarantee that this type of a graph for example, can colour every other graph, which is not 3-colourable. For example, if you look at this graph, this is not 3-colourable, but even if you try the colouring heuristic on this, it does not work because it is a fully connected graph. So, every time you introduce something on the stack your… Of course, left with just these 3 nodes; these can be coloured with 3 colours, but when we try introducing this, it will be connected to all others and therefore, we need a forth colour for this. So, this is how we do optimistic colouring.

This is the end of the lecture on Register Allocation.

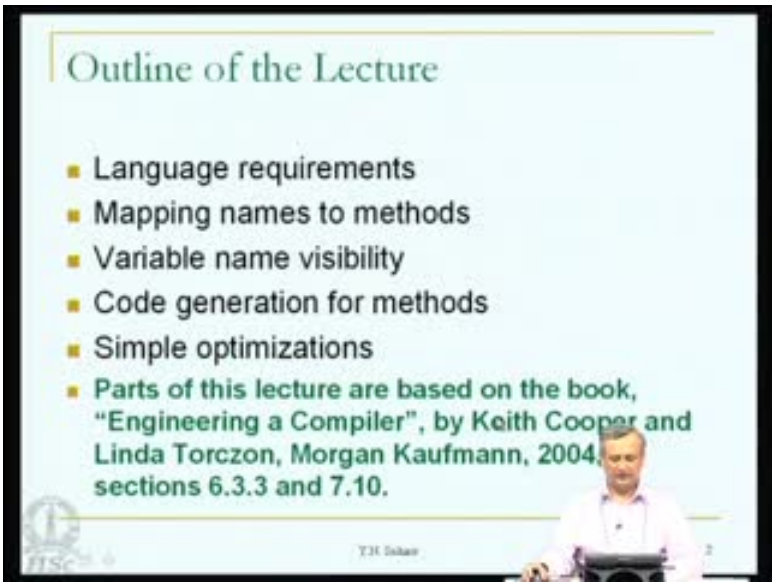
(Refer Slide Time: 19:10)



Now, we will begin the new lecture on Implementing Object-Oriented Languages.

Welcome to the lecture on Implementing Object-Oriented Languages.

(Refer Slide Time: 19:23)

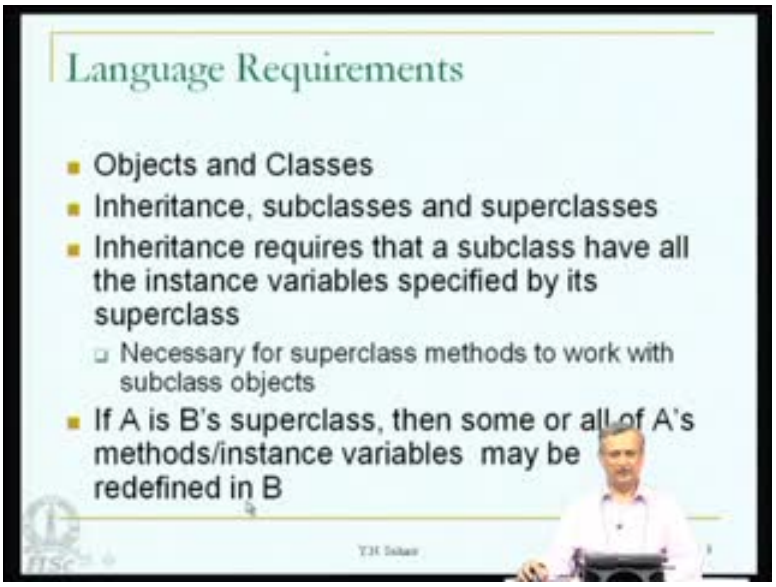


In this topic, we are going to basically consider classes and objects. Then, see – what are the requirements of the language, how to map names to methods; there are so many names in the program; it is possible that some of them are in the sub classes, some of them are in the super classes, some of them may be really fine and so on and so forth. So, how to map these names to the appropriate methods? That is something we are going to

study; then, variable name visibility, code generation for methods, and then finally, some of the simple optimizations.

I have been adopted some parts of this lecture from this book by Cooper and Torczon.

(Refer Slide Time: 20:24)

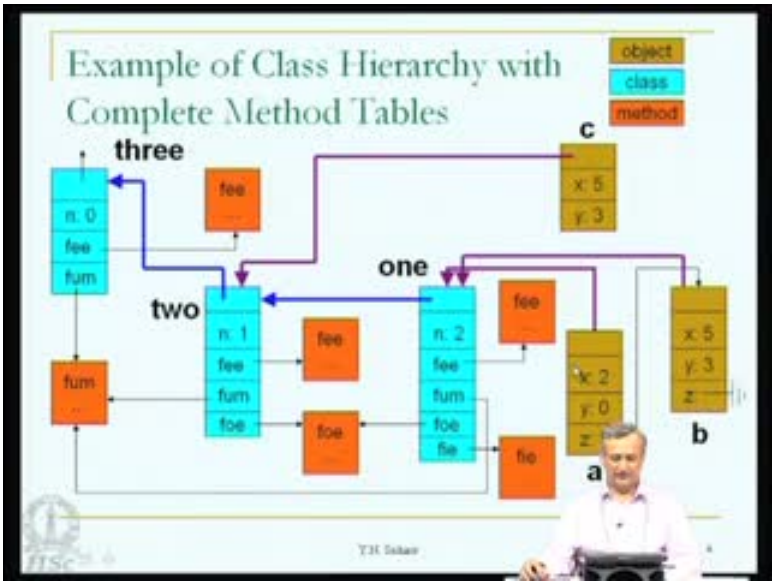


What are the language requirements? In an object-oriented language, we have objects and we have classes. For every class, you can instantiate it and create an object. So, these are all well known. Then, we have inheritance, subclasses and superclasses. So, you first declare a class, then inherit from it, and create a subclass. The class from which I have inherited is called as a superclass.

Inheritance requires that a subclass have all the instance variables specified by its superclass. In other words, if the superclass has three variables, which are actually declared, then every subclass of this superclass will have all these three variables in it. So, this is necessary for the superclass methods to work with subclass objects. As you know, you can actually have a base class pointer point to any object of the type, which is derived from this base class. If that is to be done, then the superclass methods must work with subclass objects and this requirement of having all the variables of the superclass inside the subclass becomes necessary. This will become clearer as we go along.

If A is B's superclass, then some or all of A's methods and instance variables may be redefined in B. That is not a problem; we should be able to provide for this as well.

(Refer Slide Time: 22:12)



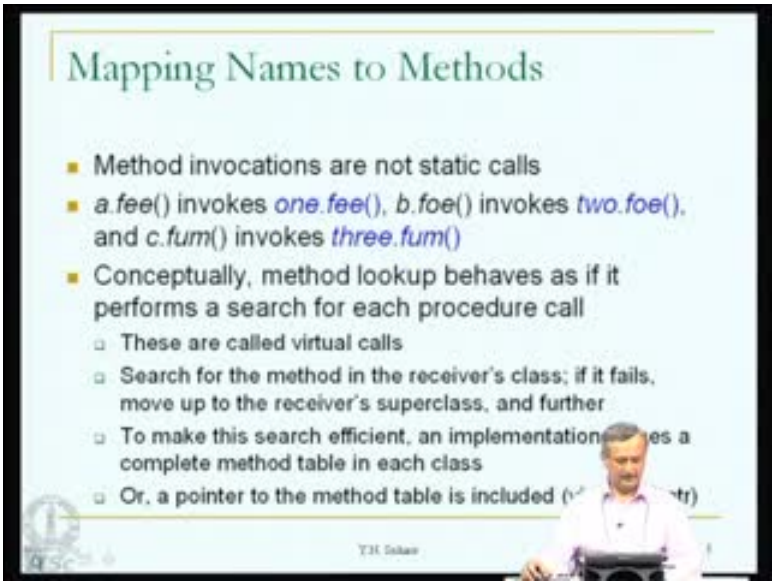
Here is a very simple class diagram. The legend is – these three in turquoise blue are classes, then the light brown are all objects, and then these orange coloured ones are all methods. So, there are three classes – one, two and three. Class one is inheriting from two and class two is inheriting from three; that is the end.

Class three, which is the grandfather class… This is the father class (Refer Slide Time: 22:53) and this is the son class. This is the hierarchy that we are talking about. It has a static variable, n, which shows how many instances of this have been created. It defines a procedure called fee and it also defines another procedure called fum.

Class two derives from three. It has one object, which has been created; that is, c. Even though it inherits fee from the superclass three, it redefines it. Now, you can see that this method pointer (Refer Slide Time: 23:35) is pointing to fee. Whereas, fum, which is inherited from three is still being reused by class two also.

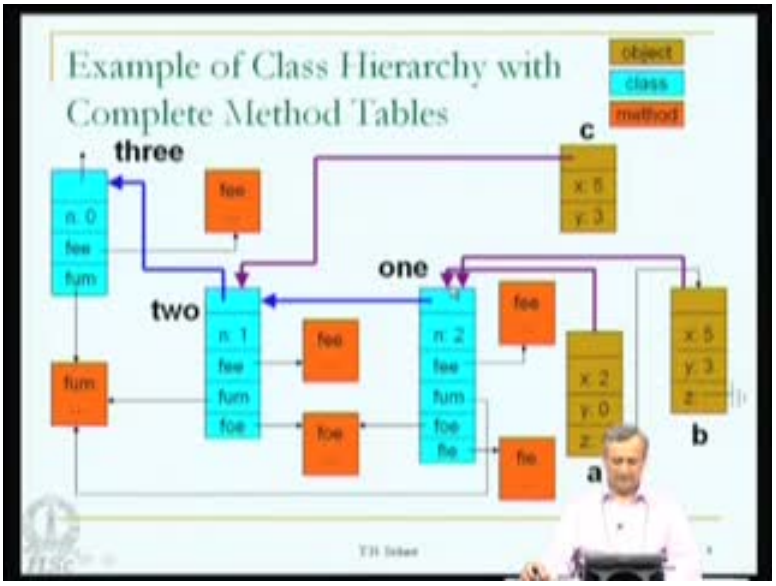
Let us look at class one. This actually inherits from two, two in turn inherits from three. So, one will have the variables of two and three both and methods also. For example, fee is inherited, but it redefines fee. fum is being reused. The same fum, which is defined by three is used here (Refer Slide Time: 24:13). foe is defined by class two and it is being reused here. fie is a new method, which is defined by class one itself. Then, there are three objects – a, b and c; a and b belong to class one, c belongs to class two, and there are no objects of type class three.

(Refer Slide Time: 24:39)



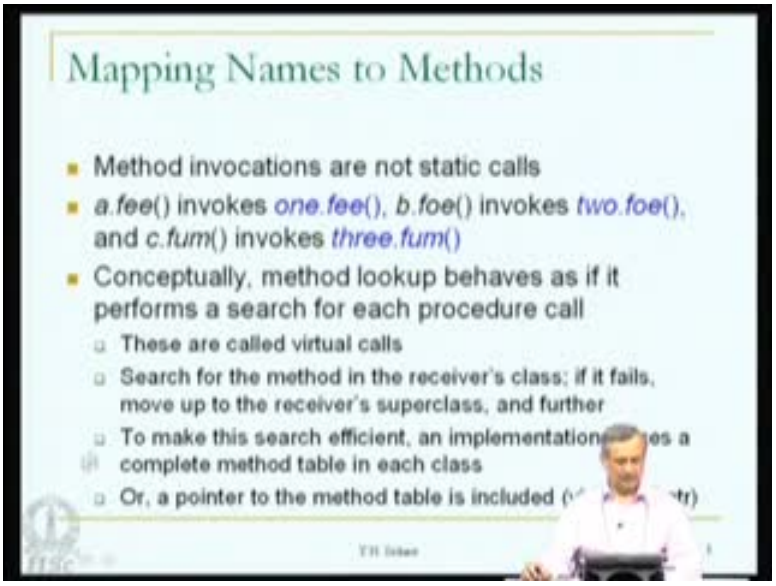
With this scenario, let us see how things work. It should be made clear that method invocations are not static calls. In other words, when we actually write a dot fee, we do not know which method is being called. This is the general statement.

(Refer Slide Time: 25:08)



In this particular case, when we say a dot fee, it invokes one dot fee. The reason is a is an object of type one and this f e e (Refer Slide Time: 25:16) invokes one dot fee, which is being redefined. Even though fee is inherited from two, it is being redefined in one. So, the local fee will be called when we say a dot fee; that is no problem.

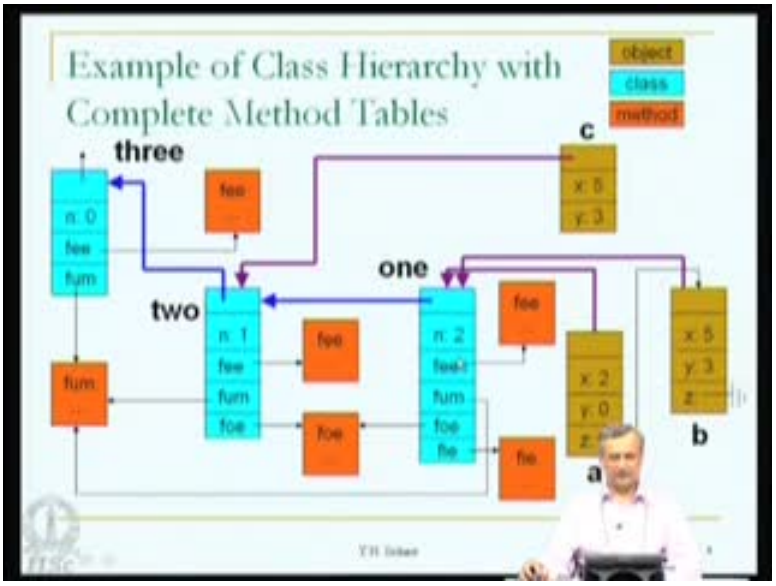
(Refer Slide Time: 25:31)



b dot foe invokes two dot foe. Let us see what happens to b dot foe. b is here (Refer Slide Time: 25:43), foe is inherited from two, and there is no local redefinition of foe for one. When we say b dot foe, it is really two dot foe, which is being called. So, there is no foe here (Refer Slide Time: 25:59). Then, c dot fum invokes three dot fum. When we say c dot fum, c is an object of class two, but it does not have its own fum, it is actually from class three (Refer Slide Time: 26:16). So, c dot fum really invokes three dot fum. This is what we mean. This is a virtual call. We have no idea of which particular call is invoked until the call is resolved.

Conceptually, method lookup behaves as if it performs a search for each procedure call. So, these are called virtual calls. How does it happen? Search for the method in the receiver's class; that is, the particular object. For example, if it is b dot foe, search in the class corresponding to b. That is, one itself. If it fails, for example, there is no b dot foe in the class corresponding to b. If it fails, move up the receiver's superclass and further. Then, the superclass of one is two. So, two defines foe. So, b dot foe resolves to two dot foe.

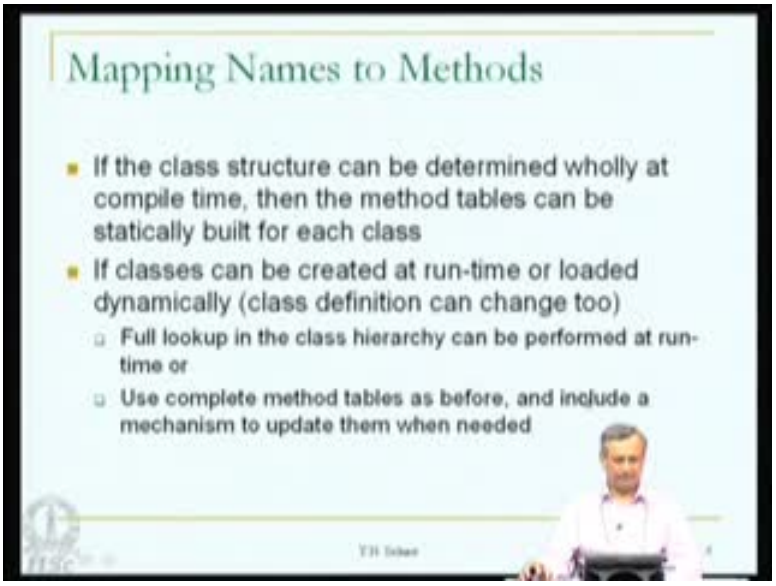
(Refer Slide Time: 27:31)



To make this search efficient, an implementation places a complete method table in each class. This is what has happened here. Each one of the classes has a method table. For example, if you consider one, there are four methods corresponding to class one: first one fee is a local redefined method. So, there is a pointer to this. This is the method table we are talking about. fum points to the old one itself, the inherited one. foe points to the old one, the inherited one. fie is a new method, which is defined within one. Same thing is similar in two also. fee is locally redefined, fum is inherited, and foe is a new one. Within three, fee and fum are both locally defined new methods.

Or, a pointer to the method table is included. So, this is called as a virtual table pointer (Refer Slide Time: 28:24). Instead of keeping this entire table inside the class, it is possible to have a table outside and a pointer replacing this entire table. So, I go to that table, which is actually this, which is outside and then call this particular method. So, that is called as a virtual table pointer.

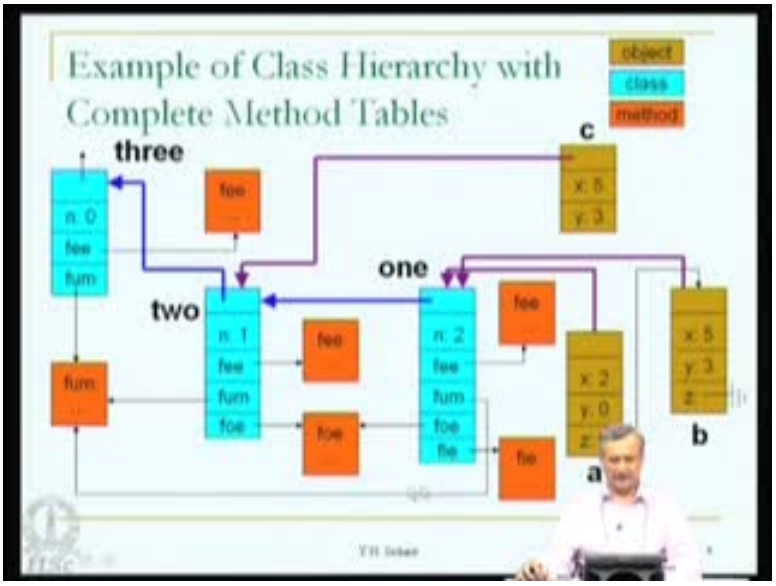
(Refer Slide Time: 28:45)



How do you map names to methods? If the class structure can be determined wholly at compile time, then the method tables can be statically built for each class. Here, in the previous example, that is how it was. Here (Refer Slide Time: 29:04), we knew which particular class is rare, which object belongs to which particular class, which methods have been defined in which class, etcetera were all known. So, we could build all the method tables statically and put them inside the class record.

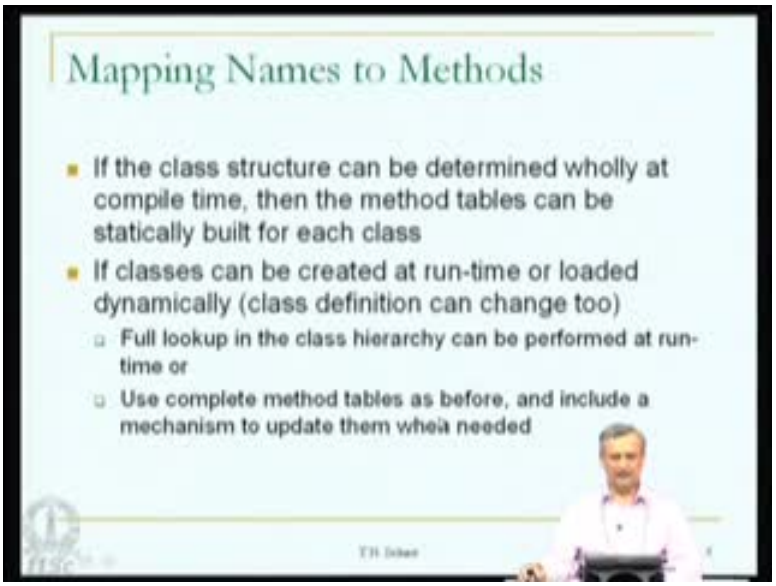
If classes can be created at run time or loaded dynamically, class definition can change as well. Like in Java, I can roll it at run time and I can also create it dynamically. Then, full lookup in the… So, if we do this (Refer Slide Time: 29:38), then there is no need to lookup, method points are all there in the method tables. So, we just go to that particular method and use it; there is no need to search.

(Refer Slide Time: 29:52)



For example, there is no need to search anything here. I have a special pointer perform right here, which can be used to call this fum.

(Refer Slide Time: 30:00)

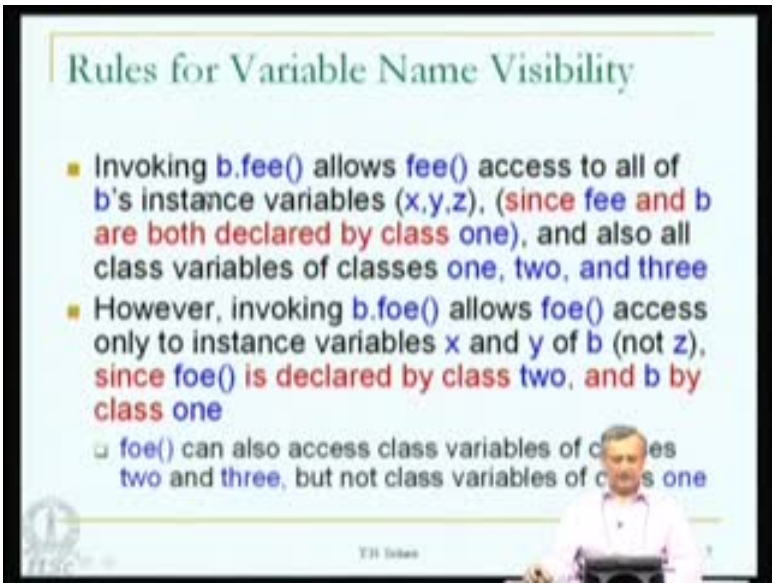


Otherwise, if it is dynamic, then full lookup in the class hierarchy can be performed at run time; no problem because if I add a class at run time, then I may not be able to build the entire table at compile time itself.

Otherwise, use complete method tables as before and include a mechanism to update them when needed. So, if we add a new class, then for all the classes actually which are

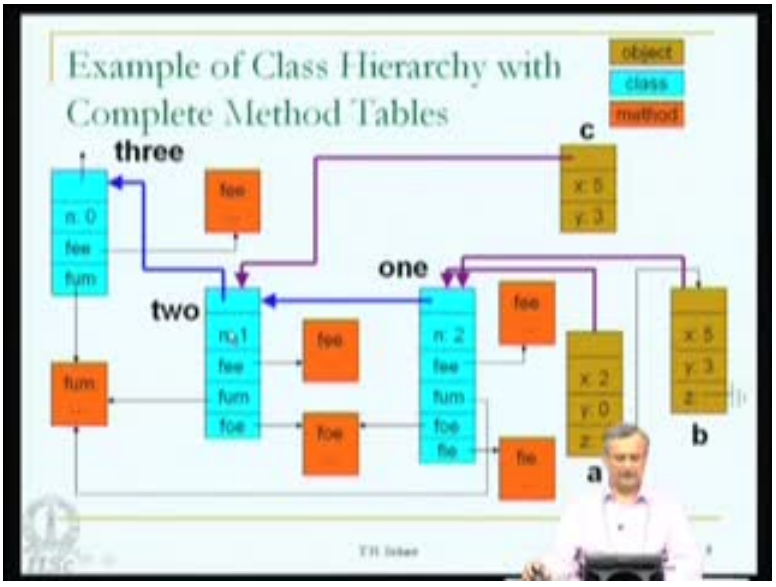
derived from this class later on, we create the method table at run time. That is what this really is.

(Refer Slide Time: 30:35)



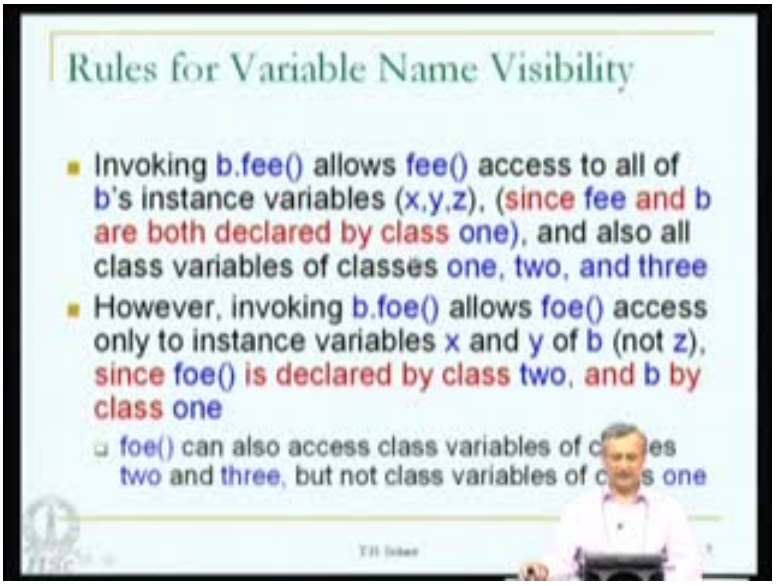
What are the rules for variable name visibility? So far, we saw the rules for searching method names. Now, let us see what happens to data.

(Refer Slide Time: 30:55)



Let us look at a picture, the same picture here. We have a and b of type class one. a has x y z, b also has x y z – these are objects of class one. Then, c has only x y and it is of type class two.

(Refer Slide Time: 31:18)



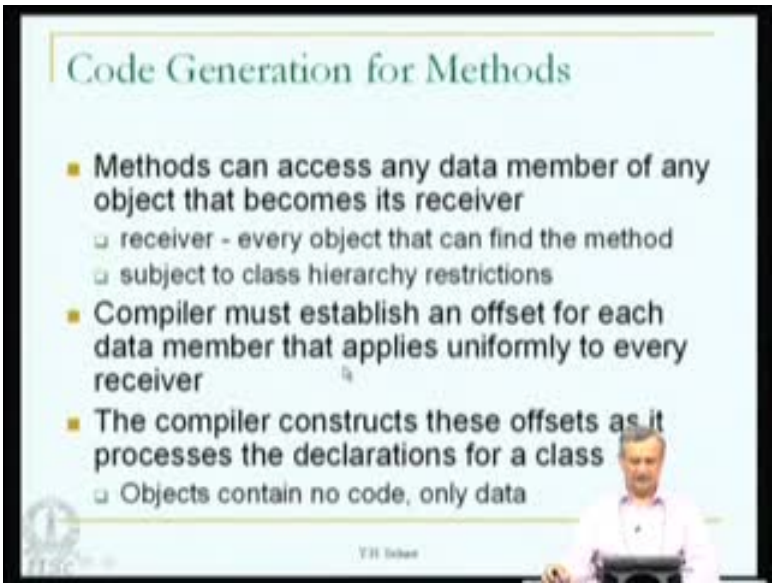
Invoking b dot fee allows fee access to all the b's instance variables; all the three of them – x y z. Why? Since fee and b are both declared by class one. So, see here (Refer Slide Time: 31:36). We call b dot fee. This fee will have access to all the variables; that is, x y z of this particular object. b is of type class one and fee is redefined right here. So, there is no problem. Everything that is declared will be accessible to the method fee and also all the class variables of one two and three. So, this is the lowest in the class hierarchy (Refer Slide Time: 32:08). So, one inherits from two and two inherits from three. So, all the variables of two and three will also be accessible to this method fee, which can actually access the variables of the superclasses.

However, when we are invoking b dot foe, there is a small problem. Let us see what happens. Here is b (Refer Slide Time: 32:32) and here is foe. foe is not defined here. foe is defined actually in class two. So, it is that foe that we are calling. When we compile two, it was not expected to work with one, but it was expected to work with two and three only. So, this creates some special problems.

Invoking b dot foe allows foe access only to instance variables x and y of b. It does not allow access to z because it never know that there is something called z. z actually belongs to one (Refer Slide Time: 33:10), the objects of class one, whereas this foe belongs to class two. So, class two has no information about class one as such. z of b is not accessible to this foe (Refer Slide Time: 33:29), since foe is declared by class two

and b is declared by class one. Foe can also access class variables of classes two and three, but not class variables of class one. So, this is what we explained just now.

(Refer Slide Time: 33:44)



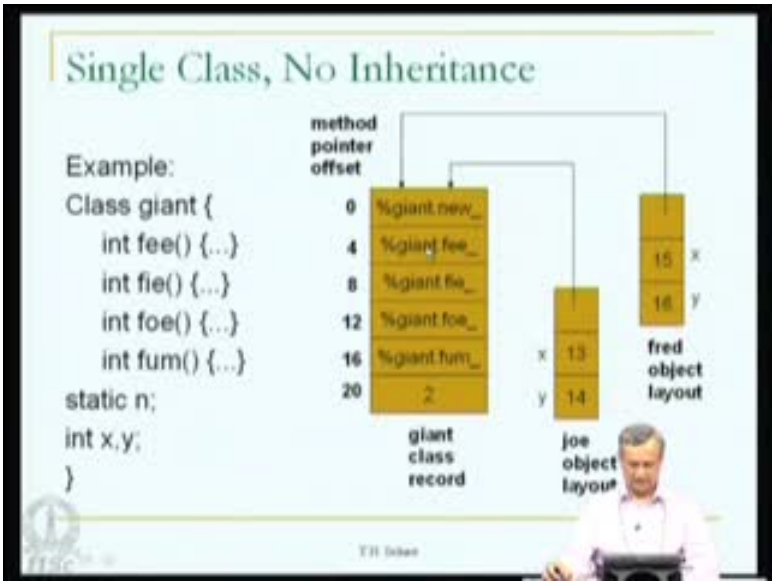
How do we generate code for methods? Methods can access any data member of any object that becomes its receiver. So, what is a receiver? Every object that can find the method is a receiver. We say b dot something and so on. That b is the receiver.

Subject to class hierarchy restrictions, whenever in the hierarchy of classes, an object can find a method and that method can be executed and methods can access any data member of any object that becomes the receiver. So, if it is b dot something and b's b dot fee, fee can access members of b.

The compiler must establish an offset for each data member that applies uniformly to every receiver. So, the problem is – if receivers are from different classes, then this creates a problem. So, you cannot access the data variables so easily. So, we will see this difficulty.

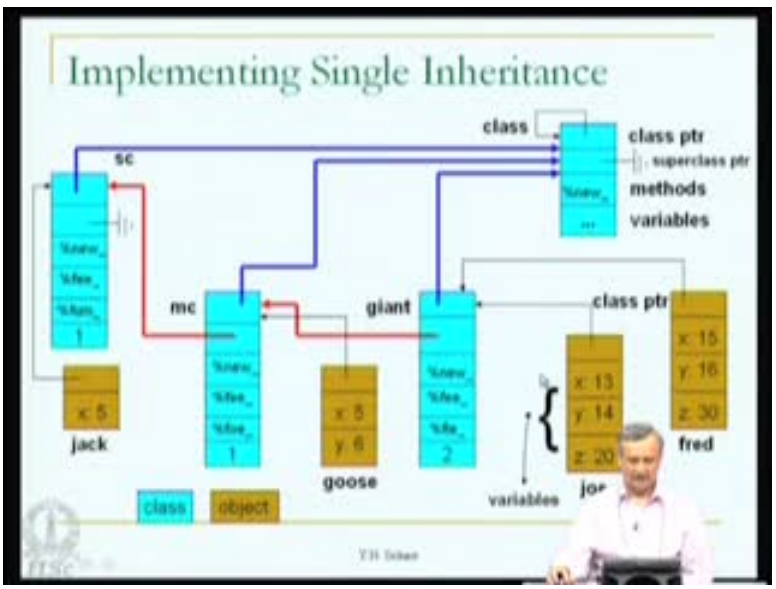
The compiler constructs these offsets as it processes the declarations for a class. So, objects contain no code, they contain only data. This is very clear. Code is actually in some other place. Let us show this for a single class with no inheritance.

(Refer Slide Time: 35:15)



Here is a very simple class – class giant. int fee, int fie, int foe, int fum – four methods. This is the static n and int x y. Objects contain variables – x y, n is within the class record and the method pointers are all here. These objects point to the class record and here are the method pointer offset – 0 4 8 12. So, when we want to call something, we know where exactly… This can be done at compile times. So, if there is joe dot fee, we know the offset of fee. So, taking the contents of this is very easy. Then, there is a giant sub routine instruction to execute this particular method.

(Refer Slide Time: 36:13)

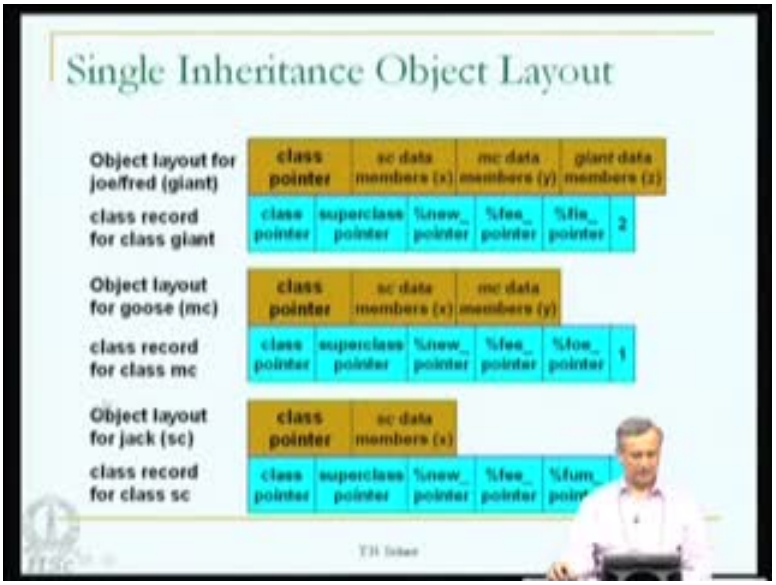


Here is the detailed class diagram with single inheritance. The previous one did not even have inheritance, here there is single inheritance. Multiple inheritance would be the last one.

We have class giant, which is of type class mc and mc itself is of type class sc. So, there is a class hierarchy of giant mc and sc. This is the superclass called class in Java, which is the mother of all classes. So, its own class pointer actually points to itself. Then, we have a superclass pointer field, then method pointer fields, and then variables. That is how all these records are laid out.

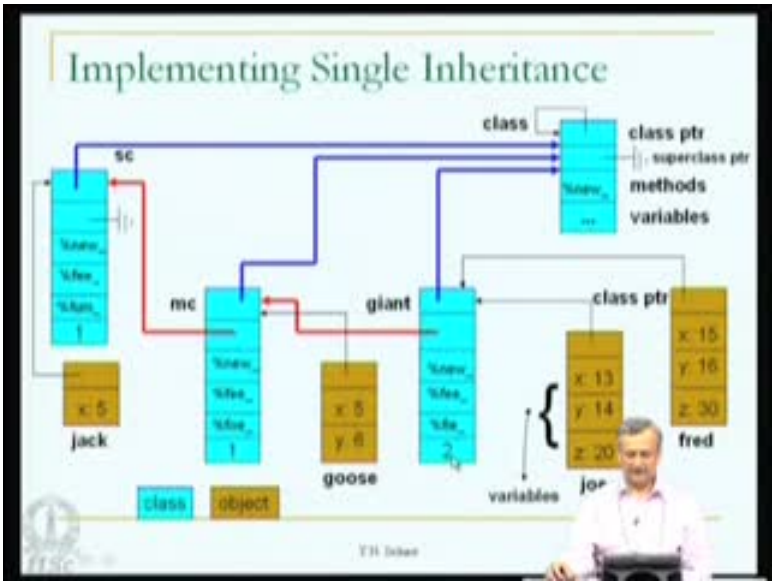
These are the variables (Refer Slide Time: 37:06) – x y and z and here are the superclass pointers. These are the class pointers. So, all of them point to this particular class itself. For objects, the class pointer points to their own class type; all these, for example, point to their own class type.

(Refer Slide Time: 37:31)



How about the object layout under single inheritance?

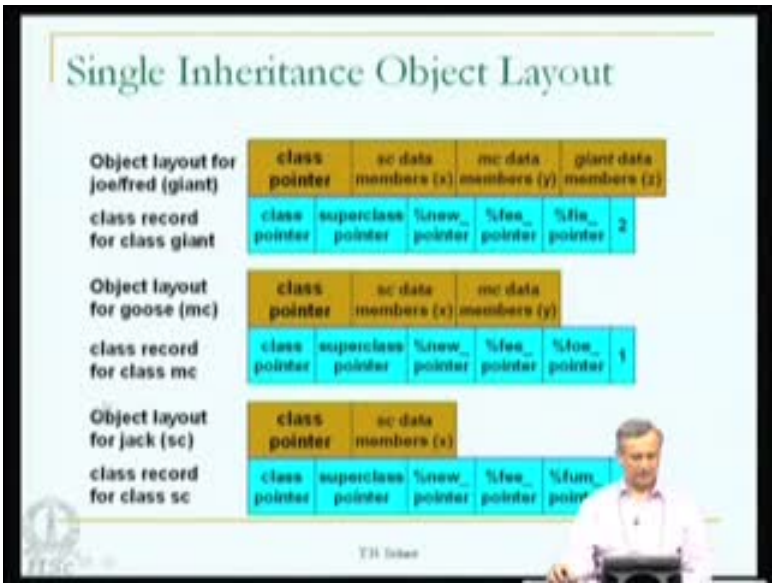
(Refer Slide Time: 37:40)



Remember this diagram. We have this giant, which has new fee and fie and then this static, which is not necessary for objects. So, objects have x y z.

For object layout for joe and fred, (Refer Slide Time: 37:56) which are members of the class giant – there is a class pointer, then the data members of sc, which is just one variable x. For example, here (Refer Slide Time: 38:10). This sc has only one variable x, this mc has x inherited from here and y its own, and giant inherits x and y from the superclasses and z is its own.

(Refer Slide Time: 38:28)

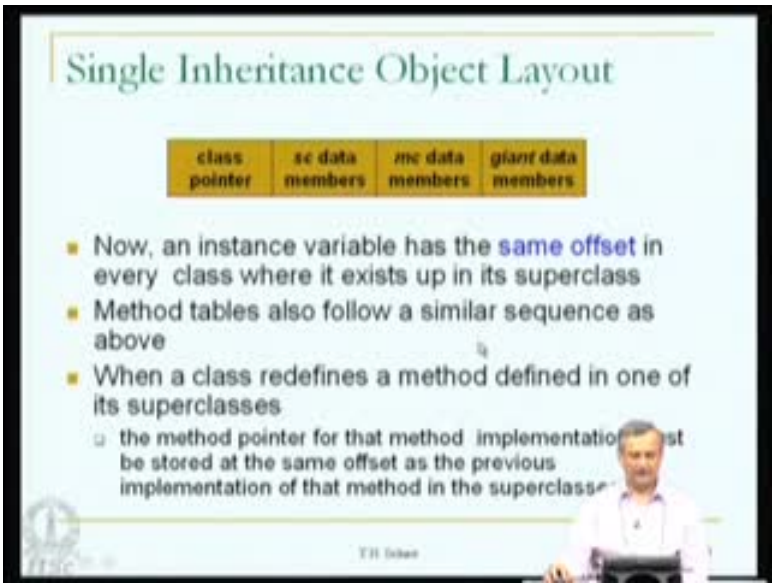


We have sc data members, mc data members and giant data members. So, this is a layout for objects of type giant. Similarly, the class record is something we already saw – class pointer, superclass pointer, method pointer table. This is the method pointer table.

Object layout for goose, which is a member of mc, which is one level above. This is mc (Refer Slide Time: 38:58) and this is the object we are talking about. It does not have the members of giant, obviously. It has superclass members sc, then data members corresponding to that, and then its mc; its own. Then, this will be of course similar (Refer Slide Time: 39:15). Object layout for sc; that is, this particular class (Refer Slide Time: 39:20), which has only x. It has only sc data members, which is just x.

Please see the order in which this data is laid out (Refer Slide Time: 39:32). When there is no inheritance, we have just our own class instance variables. When there is one level of inheritance, then we have the superclass data members followed by our own data members. When there is two levels of inheritance, we have the grandfather superclass data members, then father superclass members, and finally, our own class data members. So, this is the order in which the objects are all laid out.

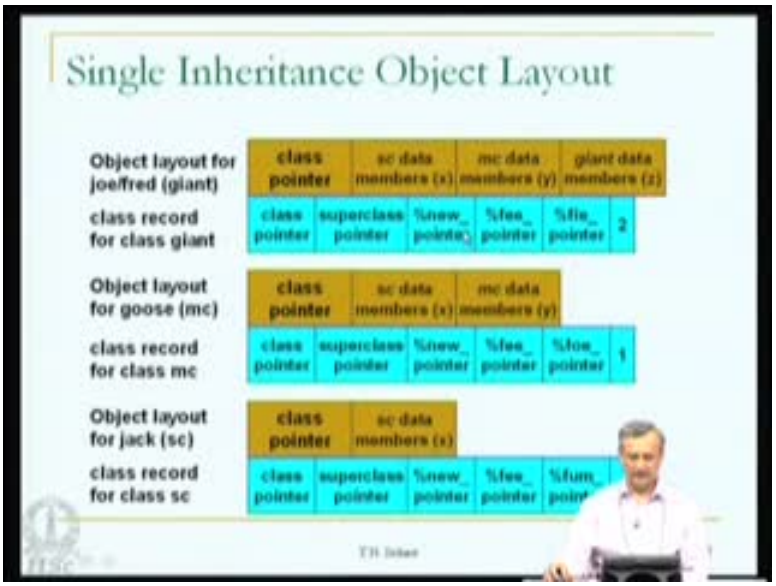
(Refer Slide Time: 40:13)



Now, the problem is - an instance variable has the same offset in every class where it exists in its superclass. So, sc will always be present in this area, mc will always be present after sc, and giant will always be present after sc and mc. So, this is the order in which the data will be laid out, irrespective of any other inheritance as well. (( )) If it was

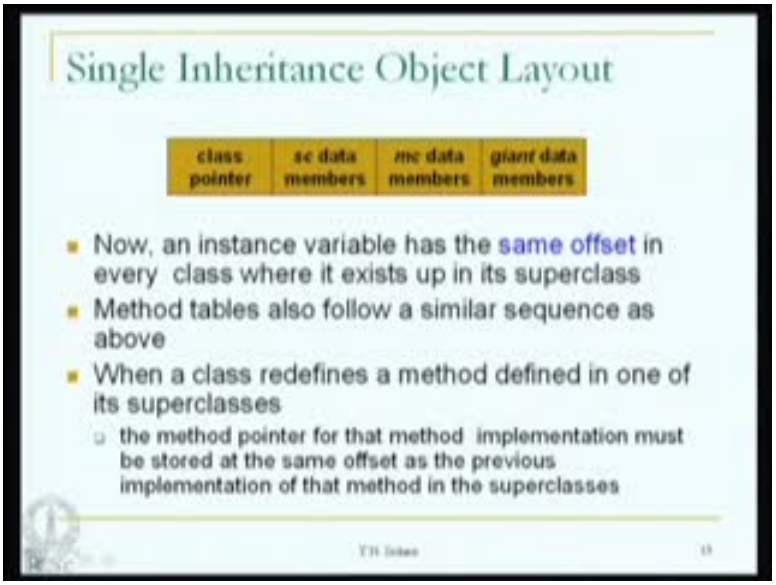
not giant, but it was some other class say – a, then we would have sc, then mc, and then a. If there were to be some other class, which inherits from giant say – b, then the data area of b would have been attached here.

(Refer Slide Time: 41:03)



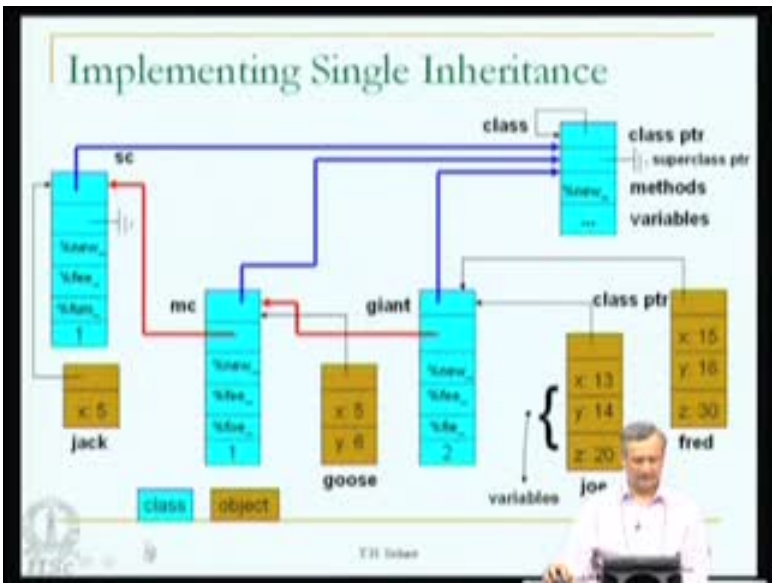
Methods tables are similar, which we already showed. This was the method table. We have the methods of the superclass. For example, if you take this, there are methods from superclass, and then our own methods, and things of that kind.

(Refer Slide Time: 41:21)



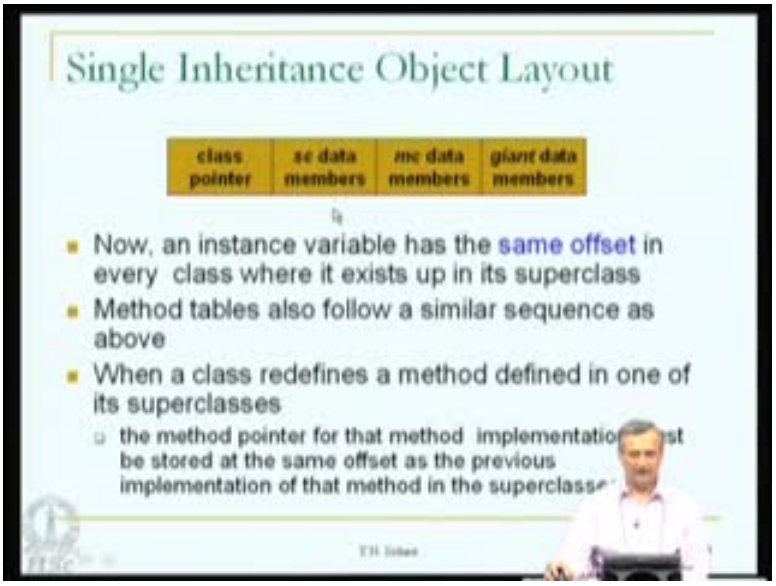
Now, when a class redefines a method defined in one of its superclasses, it is very easy, just the method pointer for that method implementation must be stored in the same offset as the previous implementation of (( )) Just replace it, nothing more than that really happens.

(Refer Slide Time: 41:41)



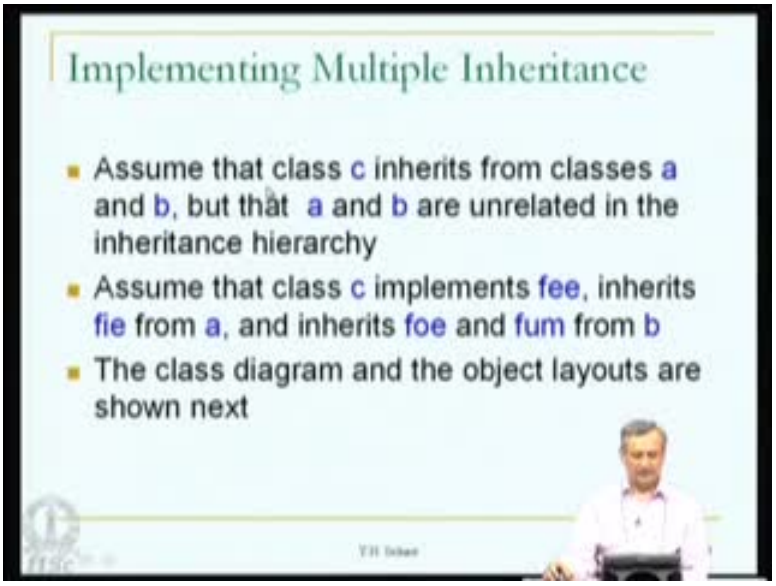
If you take this, there is no problem. For example, if fee is here, also fum is here. So, fee is being redefined here. I would possibly have a new pointer here. So, whatever fee was pointing to here will not be pointed to by this, a new method pointer will be placed in the same location. However, the order is important. So, that is a very important thing that we need to notice.

(Refer Slide Time: 42:17)



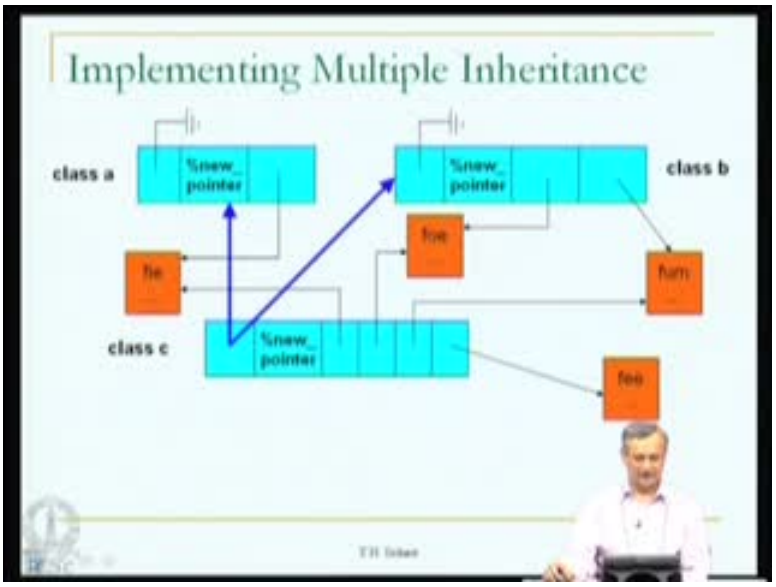
Now, what happens is – for example, when we have a method that belongs to the giant class, all these classes would have been compiled together. So, there is absolutely no problem. If we pass this pointer, all the data members would be accessed appropriately. When the method of the class mc is called, we still pass the same pointers. Since the data area is ordered with the first superclass, second superclass, third superclass, etcetera in that order, the methods can actually access these data members appropriately. There is absolutely no problem as far as accessing it. Every time we pass this particular pointer to this object and since the data layout is the same, everywhere the methods can access the variables and execute appropriately.

(Refer Slide Time: 43:22)



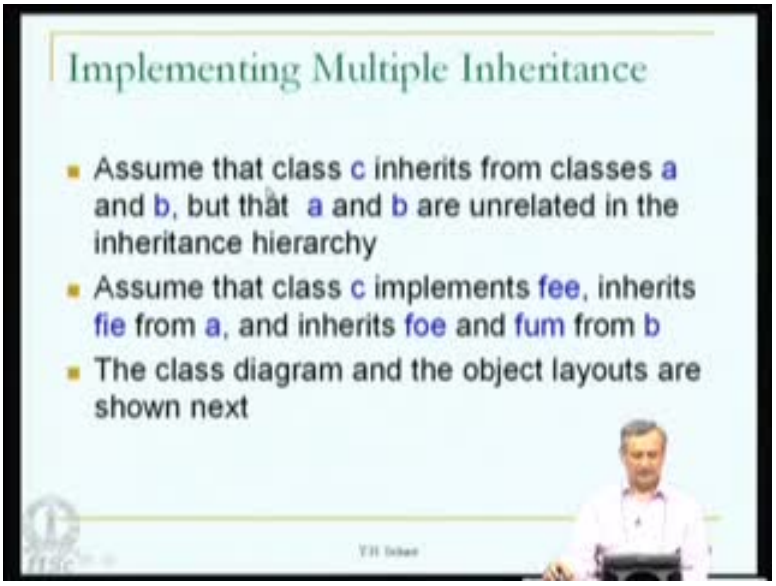
This is not the same in the case of multiple inheritance. Let us see what happens in multiple inheritance.

(Refer Slide Time: 43:31)



Here is the picture. Here is class c, it inherits from both class a and class b. So, there are two classes from which it inherits. It has a method fie, which is from class a, which has a method foe, which is from class b, and it has a method fum, which is again from class b and there is a method fee, which is its own. So, these are the method tables of the new class c.

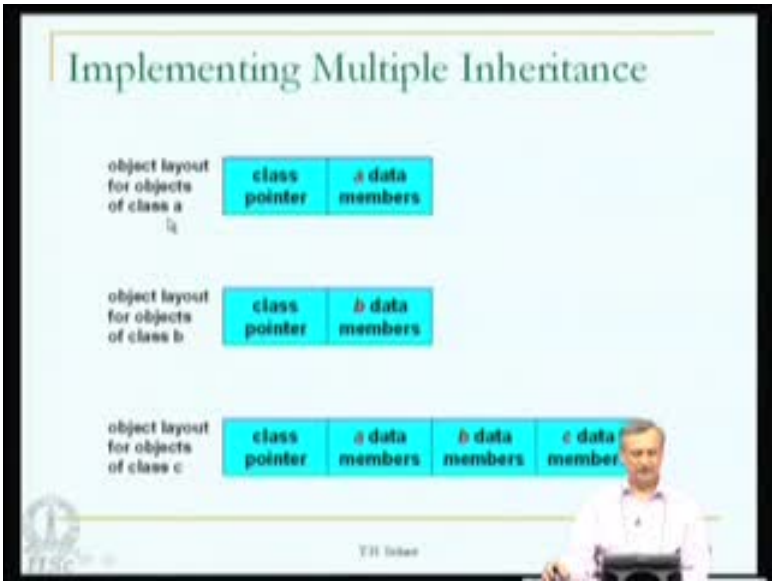
(Refer Slide Time: 44:01)



Assume that c inherits from classes a and b, but that a and b are unrelated. We saw this already. These two are not related (Refer Slide Time: 44:10).

Assume that class c implements fee, inherits fie from a, and inherits foe and fum from b. This is something I explained.

(Refer Slide Time: 44:24)



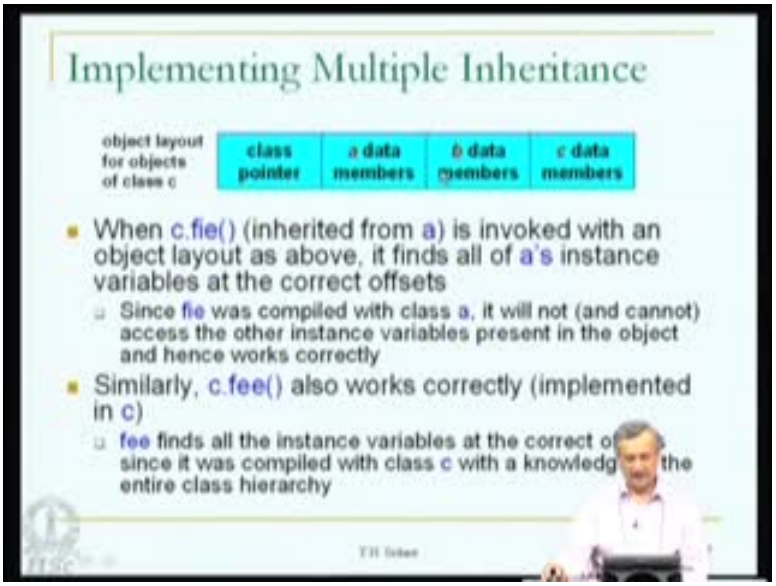
Now, look at the object layouts for that of class a. Here is class a (Refer Slide Time: 44:31), it is not inheriting from any other class. Class b is also not inheriting from any other class. Only class c inherits both from a and from b.

For class a objects, we have a class pointer and then the data members of a; no problem. For class b, which is independent and unrelated to class a, there is a class pointer and then the data members of b. The trouble starts with class c.

Class c inherits both from a and b. So, should we place a's data members first or b's data members first? c's data members will always come after the members of a and b. So, that is not an issue. Should a come first or b come first? The trouble will be – if we have a method of class c, which is inherited from a and it is called, then the pointer to this object is passed here (Refer Slide Time: 45:32). So, it will expect that the data members of a appear here. That is because, it works with the layout of objects for class a, which the members are right here. So, it expects that the members of a are right here.

However, if we call a method from c, which belongs to b, it expects similarly, the b data members immediately after the class pointer. That is, in this area (Refer Slide Time: 45:58) because that is the way it is happening in the objects of class b. However, if you place b here, then the method call of a will be inappropriate. If you place b here, then the methods call of b will be inappropriate. c will always work properly because it is always after a and b. How to solve this problem?

(Refer Slide Time: 46:23)



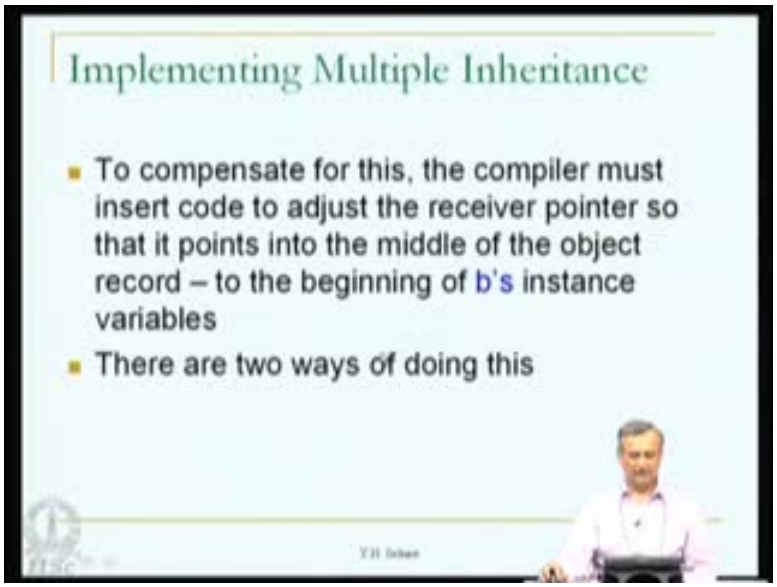
That is what is mentioned here. When c dot fie inherited from a is invoked with an object layout as here, it finds all of a's instance variables at the correct offsets. No problem at all. Since fie was compiled with class a, it will not find and cannot access the other

instance variables present in the object and hence works correctly. So, there is no problem; c dot fie will work well because all these were compiled together.

Similarly, c dot fee also works correctly implemented in c because we are assuming that a is first and then b. So, that is why this c dot fie works properly. c dot fee also works properly because everything were compiled together, a and b are placed, and then c is placed.

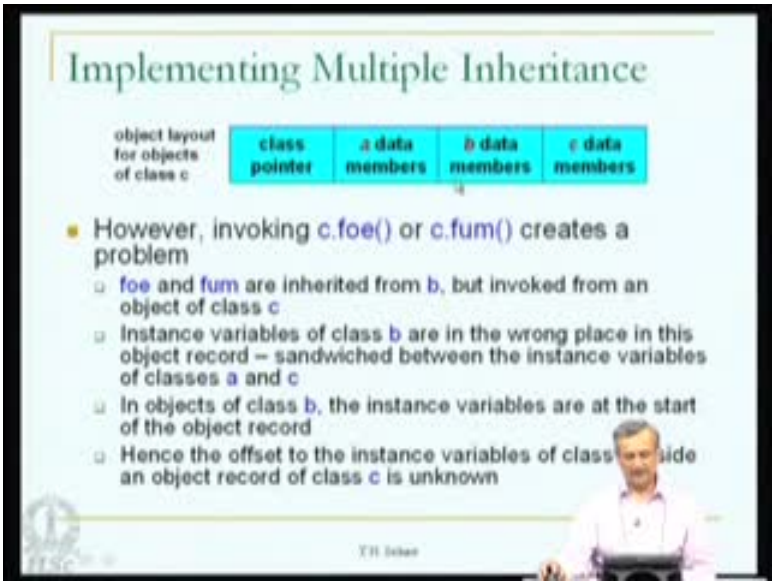
However, c dot foe or c dot fum will not be proper. So, c dot foe (Refer Slide Time: 47:21) or c dot fum are from class b. With this arrangement, as I mentioned, b is supposed to be expected here (Refer Slide Time: 47:29), but b is here. Therefore, it will not work properly. foe and fum are inherited from b, but invoked from an object of class c. Instance variables of class b are in the wrong place in this object record - sandwiched between the instance variables of classes a and c. In objects of class b, the instance variables are at the start of the object record. They should have been here (Refer Slide Time: 47:55). Hence, the offset to the instance variables of class b inside the object record of class c is unknown. So, this is a problem. Where is b?

(Refer Slide Time: 48:05)



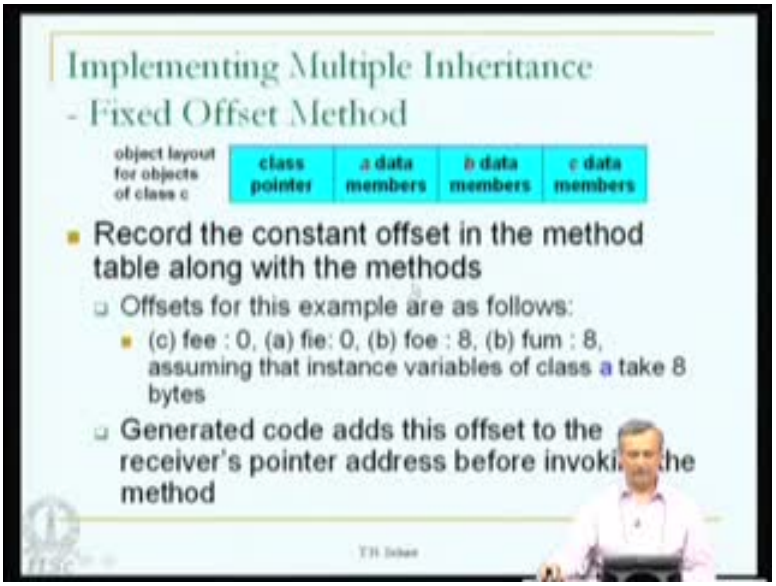
The solution to this problem is – to compensate this, for this, the compiler must insert code to adjust the receiver pointer so that it points to the middle of the object record, that is, to the beginning of b's instance variables.

(Refer Slide Time: 48:23)



What we are saying is – if we know that this is the beginning of b, which we will know. If this is the layout, we know how many data members are here. So, we will know what is the offset; this offset plus this offset will be known to us. If we make sure that we point the object pointer to this particular point inside this record and then call the methods of class b, everything will work out very well because these data members are expected to be in the beginning. So, if we point the pointer here, this will be the first set of data members and it will work correctly. How do we do this?

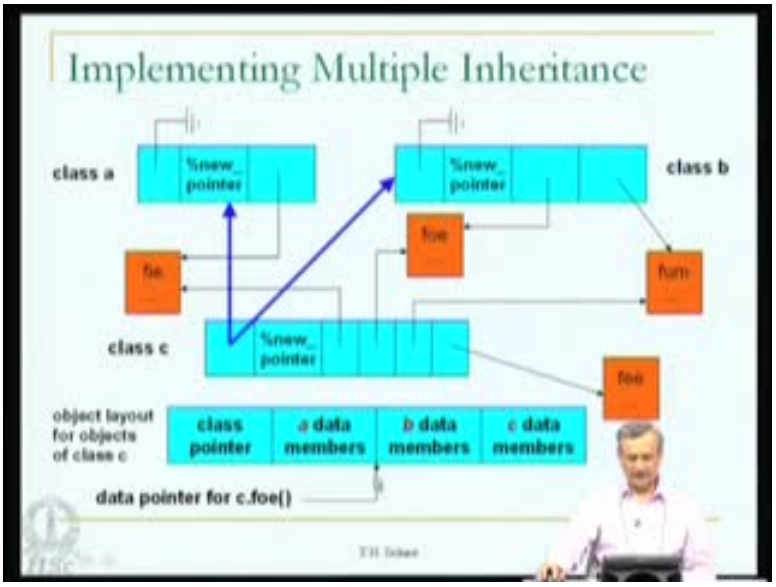
(Refer Slide Time: 49:07)



One is called the fixed offset method. Record the constant offset in the method table along with the methods. For example, for c, the offset of fee is 0 because c was compiled along with a and b. So, we can pass this pointer itself and everything will work properly. This is the extra offset. For fie, which is a's method and again the offset is 0 because a's data members are always here and a expects that they are in this place; so, nothing wrong. Only b creates a problem. For b, the offset is 8 both for foe and for fum. Assuming that the variables of class a take 8 bytes; that is, this portion requires 8 bytes (Refer Slide Time: 50:01). That is the assumption.

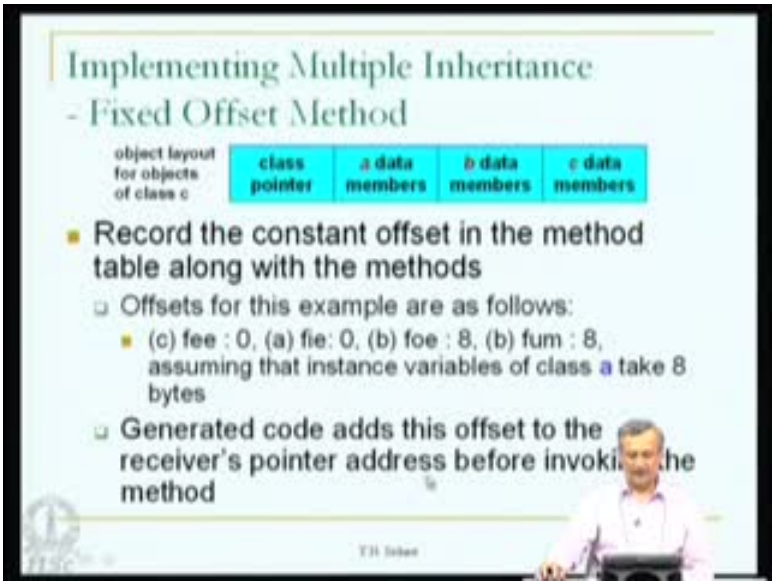
The generated code adds this offset to the receiver's pointer address before invoking the method. So, if you are pointing here (Refer Slide Time: 50:10), we add this and then it will point here. Now, everything works well.

(Refer Slide Time: 50:15)



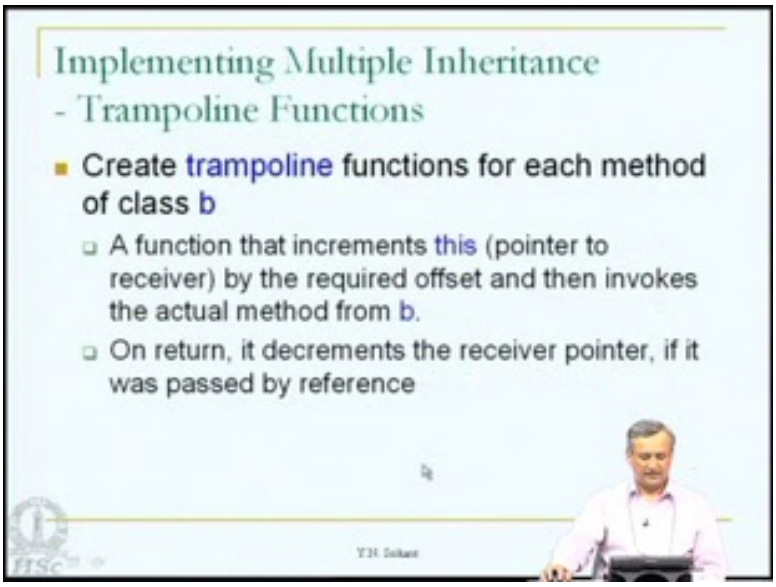
That is, data pointer for c dot foe is pointing to this point; otherwise, it will have pointed to this and then we would have had a problem. So, this was actually adding this offset.

(Refer Slide Time: 50:33)



Whatever offset I showed here, this 8 was added and instruction was generated to add this offset to the receiver's pointer address and then call the method.

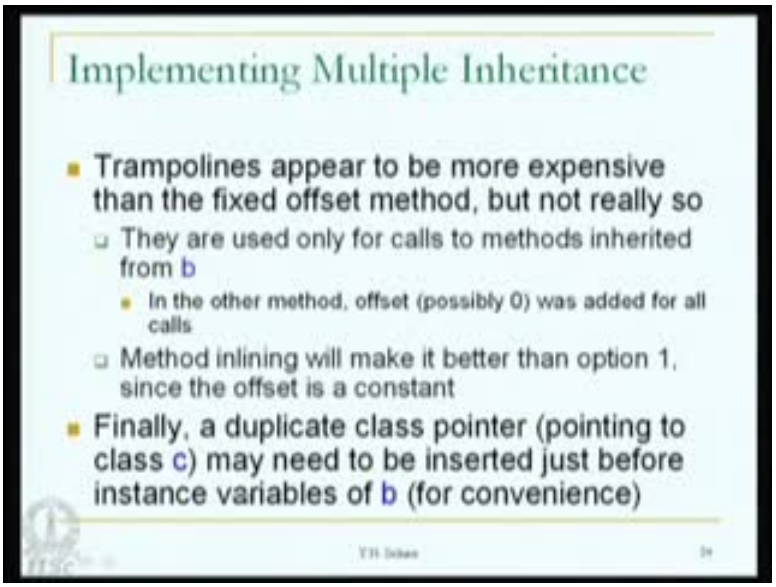
(Refer Slide Time: 50:42)



The other is what is known as a trampoline function. We create a trampoline function for each method of class b. For example, a function that increments this pointer that is the pointer to the receiver by the required offset and then invokes the actual method from b. So, we were doing it by an explicit instruction, but now, we have a special function that increments the pointer by the required amount and then invokes the actual method from

b. So, this is just using an extra function. On return, it decrements the receiver pointer if it was passed by reference and then gets back.

(Refer Slide Time: 51:32)

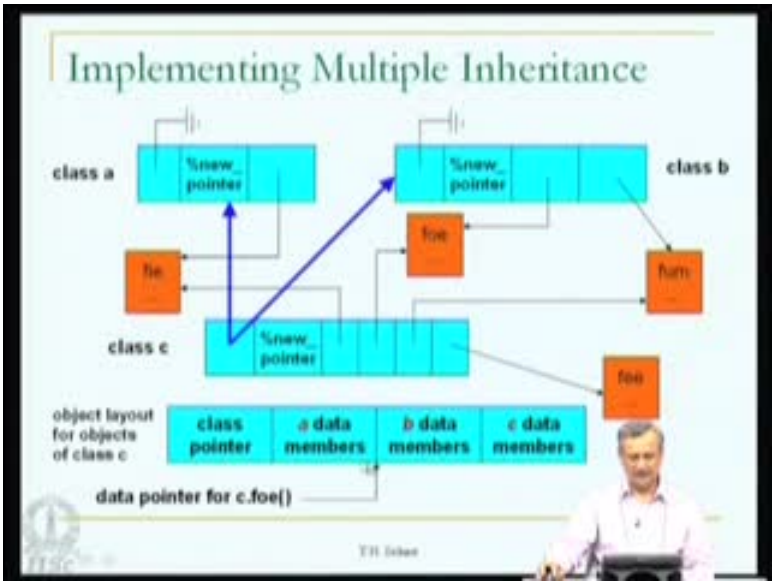


Now, you may wonder whether this is efficient at all. Trampolines seem to appear to be more expensive than the fixed offset method. Is that so? However, not really so. For example, they are used only for calls to method inherited from b, whereas the other method – fixed offset method, the offset possibly 0 was added for all calls. Whether it was from a or from c, the offset was added, you added a 0 and then that instruction was any way executed, whereas here, we do it only for the methods of b.

Method inlining will make it better than option 1, since the offset is a constant. So, we do not have to take it out from any table like in the previous case, this is actually a constant. So, inlining will make sure that you use a constant addressing mode, which you can utilize a constant operand. Since it is used only for the methods of b, this may be quite efficient.

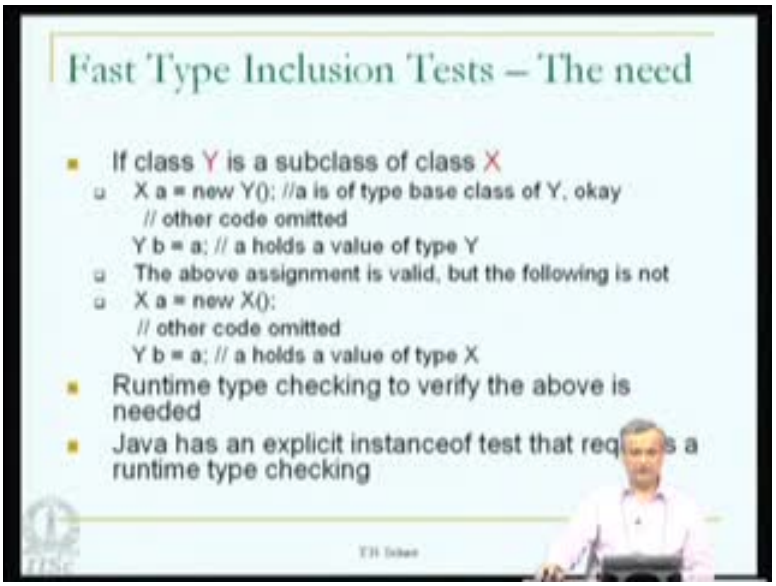
A duplicate class pointer pointing to class c may need to be inserted just before the instance variables of b. Let me show why?

(Refer Slide Time: 52:45)



For example, if we had been pointing to this place when we have class to the methods of a or c, what we really need to do is – when we call the methods of b, we insert this class pointer here. We insert this here and then point it to the beginning of the class pointer and call it so that it looks exactly like this. It is as if we are calling something of an object of class b is being passed on. So, this is more for convenience not exactly essential for the operation.

(Refer Slide Time: 53:27)



At this point, we will close this lecture and in the next lecture, we will look at some optimizations, which can be performed on object-oriented language programs.

Thank you.