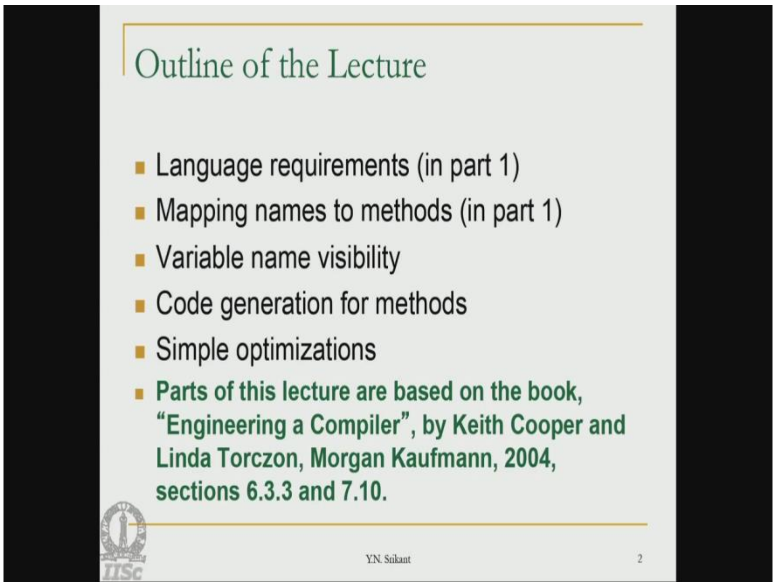
Principles of Compiler Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 28
Implementing object-oriented languages Part - 2
Global register allocation Part -1

(Refer Slide Time: 00:23)



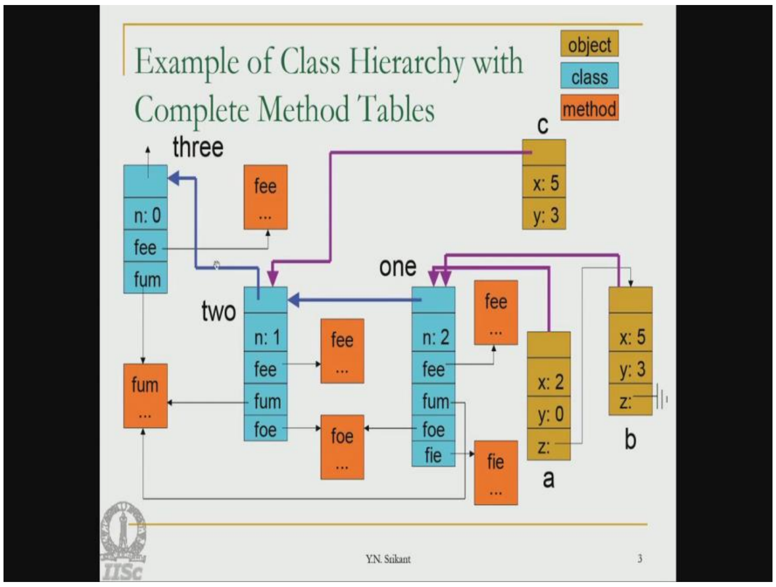
Outline of the Lecture

- Language requirements (in part 1)
- Mapping names to methods (in part 1)
- Variable name visibility
- Code generation for methods
- Simple optimizations
- Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.

Y.N. Srikant 2

Welcome to part 2 of the lecture on Object Oriented Languages. Today, we will continue with our discussion on variable name visibility, mapping method, names to methods, and so on and so forth.

(Refer Slide Time: 00:35)



Example of Class Hierarchy with Complete Method Tables

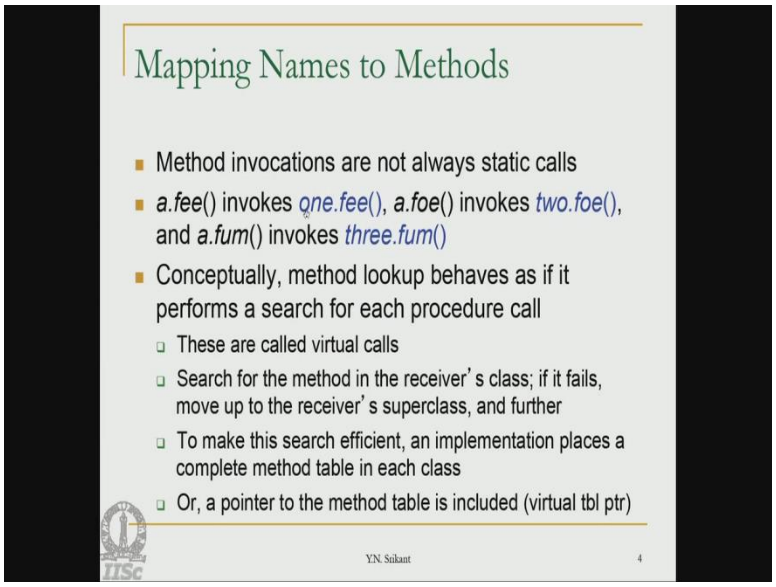
To do a bit of recap this is the diagram with class hierarchy that I showed you last time. So, here there are three classes 1, 2 and 3, the class number 3 is at the highest level in the hierarchy two derives from 3, and one derives from 2, so this is the way hierarchy is maintained. Now, there are methods that three defines for example, it defines the method fee and it also defines the method fum, fee is not shared between you know 3, 2 and 1.

Two has it is own procedure method called fee and a similarly one also has it is own method called fee, so that part of it is not shared, whereas the method fum which is available in class three is shared between 2 and 1 both. So, the class 2 does not define its own form of fum similarly class 1 also does not define its own form of fum. And similarly a method four is not a member of class 3, but it is a member of class 2 and it is shared between 2 and 1 it is provided by class 2 and it is used by class 1, so this is the difference as far as the methods are concerned.

So, similarly we also have class one which defines fie actually fie which is it shown function and it is not actually available to any other class as well. Now, there are objects a, b and c, so the object c is of type two and it has two fields x and y, object a and object b both are both of type one, they have, three fields x y and z. So, the most important thing here is to find out, which name maps to which method, when a method call is made. And similarly whenever we use a field name can a particular method actually access that field at all, so the point here is if we actually take the you know object of type one, that is object a.

And we try operating with the method four, which is actually defined in class 2, then the the field z of this object will not be available to this method four, because any object of type class 2 will not have any object you know field z inside it, it will have only x and y. So, x and y will be visible to the method four, but z will not be visible to the method four, but the z will be visible to actually the method fee, you know this here, we have method fie and here we have method fee ((Refer Time: 04:15)). So, both of them will be able to access this particular field z of, but I know object a. So, there are visibility rules of this kind and then the naming conventions as well.

(Refer Slide Time: 04:30)



Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee*() invokes *one.fee*(), *a.foe*() invokes *two.foe*(), and *a.fum*() invokes *three.fum*()
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

Y.N. Srikant 4

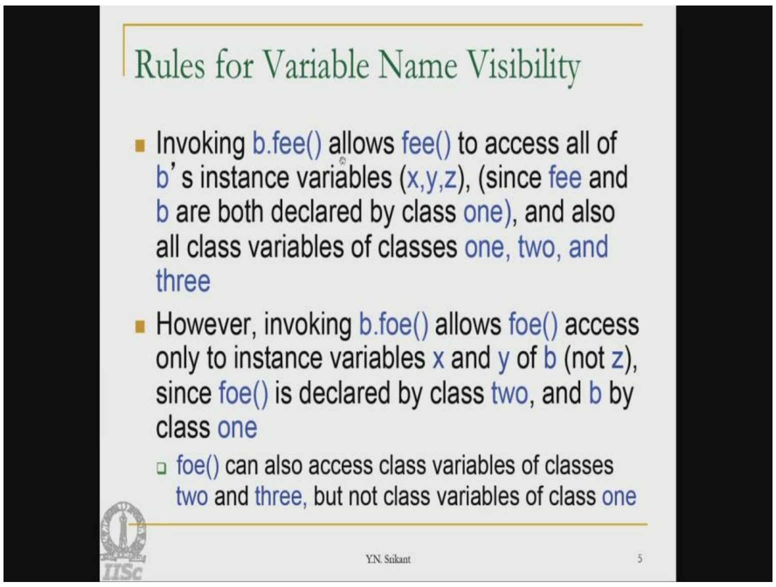
So, mapping names to methods implies finding out which particular method that the name corresponds to. So, if we say we are calling a dot fee, then does it correspond to you know one dot fee or two dot fee or three dot fee; obviously, when it is object of type you know three this thing one, that a belongs to then it would be one dot fee; if the call was made from an object which corresponds to class 2, then it would be two dot fee. And similarly if it is made from an object of the type class three then it would be you know three dot fee.

And then what about a name such as a dot four, a is the object and we are trying to make a call foe. So, foe is defined if you remember ((Refer Time: 05:35)) foe is defined only in class two and it is shared between one and two. So, foe invokes two dot foe always. So, if an a dot fum, similarly invokes three dot fum, because there is the only class which defines this method fum, so this the difference.

So, conceptually when we want to determine the method which corresponds to a particular name the method look up actually is nothing but a search for the name of that particular procedure call. So, these are actually called virtual calls and search for that method in the objects class that is the receivers class from which it was made.

You know the first of all we check whether it is a local method available in the local class itself and if that fails we go to the parent of this particular class find out it if the method belongs to super class and so on and so forth. So, this method may become very efficient provided the complete method table is stored in each one of the classes, so I will you know show you what I mean very soon. So, if the complete method is table is not stored in each class then the search may take a little extra time, so that is the difference between you know the search strategies.

(Refer Slide Time: 07:12)



Rules for Variable Name Visibility

- Invoking b.fee() allows fee() to access all of b's instance variables (x,y,z), (since fee and b are both declared by class one), and also all class variables of classes one, two, and three
- However, invoking b.foe() allows foe() access only to instance variables x and y of b (not z), since foe() is declared by class two, and b by class one
  - foe() can also access class variables of classes two and three, but not class variables of class one

Y.N. Srikant 5

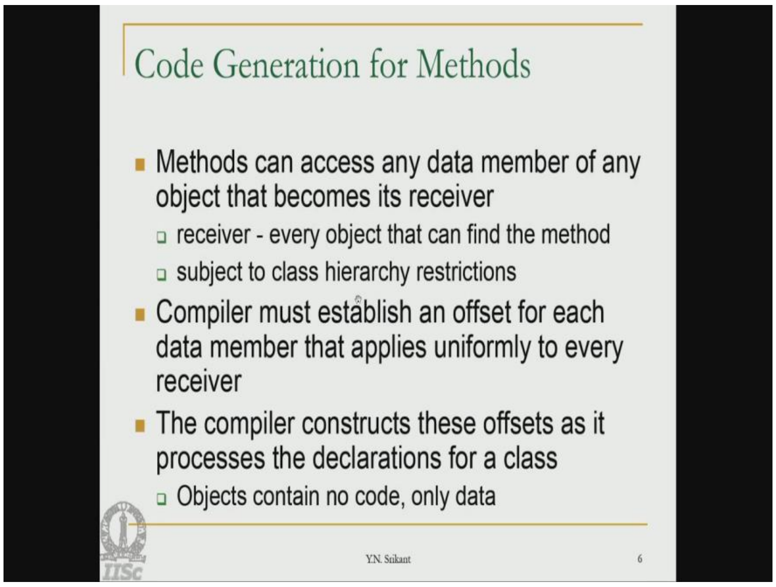
Now, the next one is variable name visibility, so suppose we call the method b dot fee, so fee is defined in each one of the classes. So, there is a local fee which is available in the classes one two and three all of them, so it allows fee to access all the variables instance variables of b, so there is no problem, because it is a local you know method that is defined inside the receiver class. So, there are three variables x, y and z, that I already showed you this diagram, so there three variables x y and z corresponding to every instance variables of class one right.

So, the class variables you know of one, two and three they actually have different types of variables. So, class one of course, has only x and y, two and three both have x, y and z. So, depending on which method is actually invoked the whether the particular variable is visible or not will be known. So, in the same breadth if we call b dot foe instead of b dot fee, then b dot foe is actually you know, so let us look at this ((Refer Time: 08:54)). So, b is here, so b is defined as that of class two. So, this is our a and b right, class one, so b belongs to class one, so that actually has three methods of the fields x, y and z and if whereas, the method foe belongs to class two.

So, anything from class two will be able to see only x and y it cannot see z, because every object of class two has only two fields x and y. So, z is invisible to method foe, whereas it is definitely available to the two methods fee and fie. So, this is that too when fee is called from an object of type class one not from an object of type class two, if the method fee is called from an object of class two then you know it can again see only x and y.

So, the moral of the story is whenever there is a method call you have to see which class that method really rather the object that are using right that point belongs to. And then determine the instance variables which are relevant to that particular class and then say these are the one’s, which are visible at that point. So, foe can also access class variables of classes two and three, but not the class variables of class one, so this is the difference. So, foe which is declared by class one you know rather class two cannot access anything, but the variables which are defined in the in it is own class, so this is something that one has to remember very carefully.

(Refer Slide Time: 10:44)



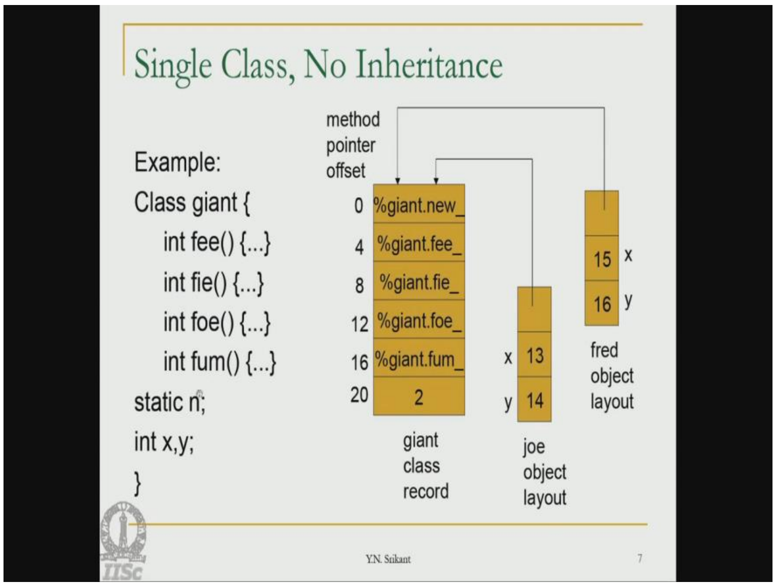
Code Generation for Methods

- Methods can access any data member of any object that becomes its receiver
  - receiver - every object that can find the method
  - subject to class hierarchy restrictions
- Compiler must establish an offset for each data member that applies uniformly to every receiver
- The compiler constructs these offsets as it processes the declarations for a class
  - Objects contain no code, only data

Y.N. Srikant 6

Now, that is about visibility, so what is visible to each method right and now how do we generate machine code for the methods. So, methods can access any data member of any object that becomes it is receiver, what is a receiver? Every object that can find the method. So of course, everything is subject to class hierarchy restrictions, which I already mentioned, now there is a basic requirement from the compiler. The compiler must be able to establish an offset for each data member that applies uniformly to every receiver, this sentence will be very clear very soon. The basic difficulty is there are several methods that can use the you know fields within an object the methods could have been in the parents class or in the current class. So, the offsets that applied to the parent class or the current class must all be the same that is what this really says. The compiler can construct these offsets as it processes declarations for a class objects contain only data they do not contain any code.

(Refer Slide Time: 12:06)



Single Class, No Inheritance

Example:

```
Class giant {
    int fee() {...}
    int fie() {...}
    int foe() {...}
    int fum() {...}
static n;
int x,y;
}
```

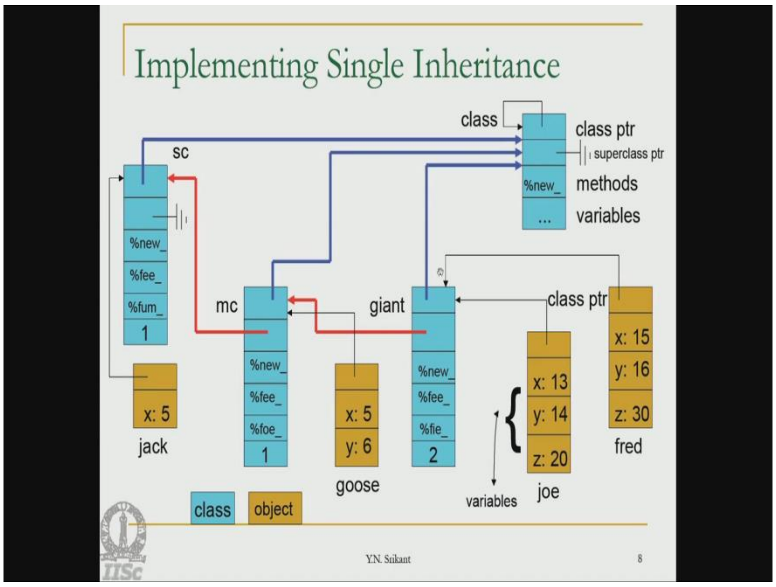
So, let us consider the cases one by one, the first one is no inheritance, so in other words there is a single class there is no inheritance and here is a very simple example class giant, so it has fee, fie, foe and fum. So, all the methods are actually not necessarily defined within this class, but here we are assuming that there is no inheritance, so all these methods are actually defined within this class, so they are not inherited from any other class.

Then there is a static variable called n and then two integer instance variables x and y, so the record for the class, so this is called as the class record name of the class is giant. So, giant class record it will have a field for the static variable and then it also has pointers to the code for each one of the methods that it has. So, fum, foe, fie and fie and fee of course, this new is a is something that is available everywhere, so because we need to create a new objects of that particular class that is provided by the system.

So, the object layout will have two fields x and y, so here also we can see two fields x and y and there is a field which points to the class record, so that is you know the one of the other fields as well. So, it is very simple whenever there is a single class and there is object let us say joe, so if we say joe dot foe automatically the compiler can you know rather at runtime, we can the compiler would have generated code for the you know offset of this giant dot foe to be accessed and then the call to giant dot for to be made appropriately.

The offset is actually static it it has been determined by the compiler because this is the method table that has constructed by the compiler, so there is no resolution left you know for the runtime at all. So, nothing is dynamically run it is also statically done the compiler simply has to take the address of the class record the beginning of the class record from the object then it knows the offset of the method foe. And it adds there to beginning of the class record, then you know the pointer gives you the beginning of the code for the method foe and a call is made to that particular method. So, it is fairly straight forward process as far as you know single class and no inheritance is concerned.

(Refer Slide Time: 15:14)

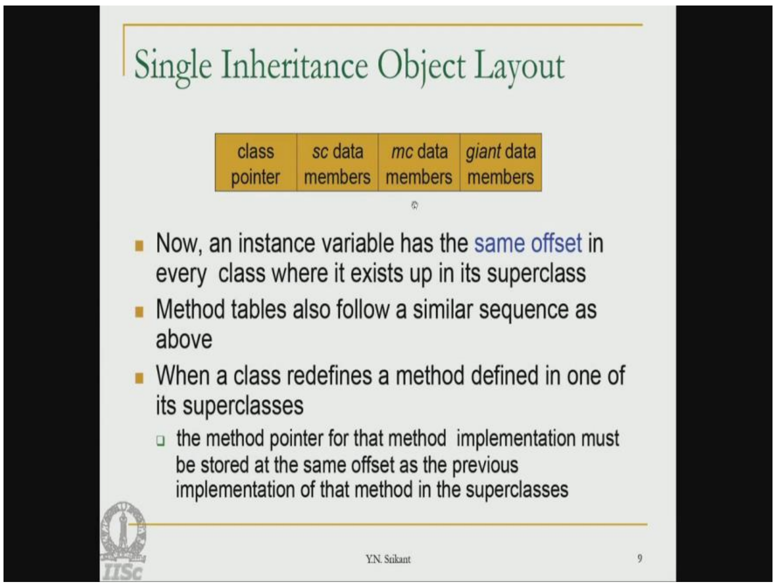


What happens if we have inheritance, the situation is slightly more complicated. So, again we have a similar diagram there is a super class called s c, there is another derived class called m c and there is a third class called giant, which is derived from m c. So, here is you know a super class record to which all classes actually have a pointer, so this contains some extra data and you know about the methods and other variables etcetera, etcetera, which are required for maintaining the entire systems.

So, let us not worry too much about this part of it these are language specific details, we need to worry about the method tables here. So, here we have the method new is available in all the classes. So, there is a pointer to the code for method, then the method fee is defined by s c, so there is a pointer to fee, the method fum is also defined by s c, so there is a pointer to fee the method fum is also defined by s c. So, there is a pointer to the code for fum there is a static variable and that is present in the class record itself.

The super class pointer is set to null here for s c, because there is no other super class above it, whereas for m c it points to s c, so here also we have the complete you know a method tables. So, m c defines of course, new, new is already provided by the system, then we have fee and then we have foe all right. So, these are the only three which are provided by this particular class, whereas fum is provided by s c. Similarly giant provides new which is already available by these provided by the system it provides fee and it also provides fie, rest are provided by foe as such. So, then there are variables x y and z for each of the objects, which are of type giant and if there is an object of type m c then it would have only two fields x and y.

(Refer Slide Time: 17:42)



Single Inheritance Object Layout

class pointer	*sc* data members	*mc* data members	*giant* data members

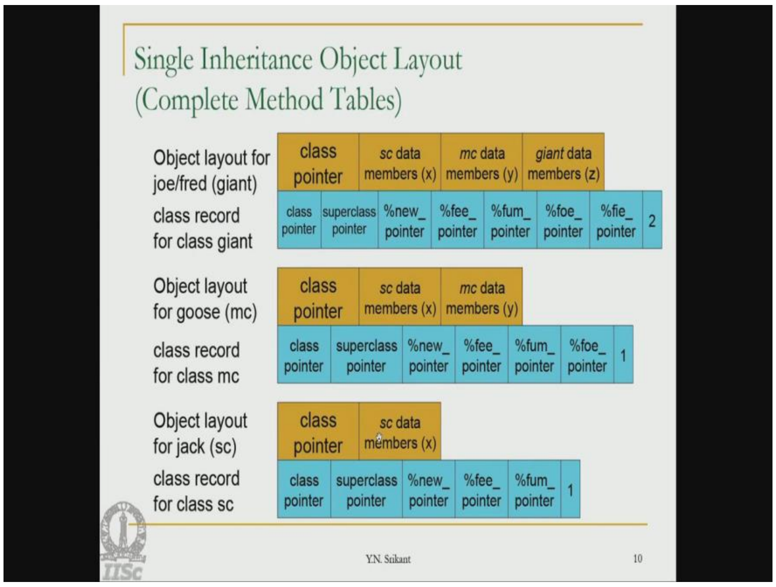
- Now, an instance variable has the same offset in every class where it exists up in its superclass
- Method tables also follow a similar sequence as above
- When a class redefines a method defined in one of its superclasses
  - the method pointer for that method implementation must be stored at the same offset as the previous implementation of that method in the superclasses

Y.N. Srikant 9

So, let us see how to generate code for this type of inheritance hierarchy, the basic idea is very simple, there is a class record pointer in each one of the objects, then there is some for the objects of the you know highest super class that is s c, because the if you consider an object of the class, which is at the lowest level of the hierarchy that is giant in our case. Then giant any data object of type giant must accommodate data members of giant, which are very specific to that particular class it derives from m c.

So, it must provide for space for the data members of m c and m c in turn derives from s c, so giant actually transitively derives from s c, so the data members of s c must also be provided for by the object layout. So, there are several components in the object layout whenever there is inheritance, they are for each one of the super classes in the hierarchy the some data area has to be set apart in the object layout of the object. So, every instance variable has the same offset in every class, where it exists up in it is super class, so this becomes clear in the picture that I am going to show you now.

(Refer Slide Time: 19:19)


Single Inheritance Object Layout (Complete Method Tables)

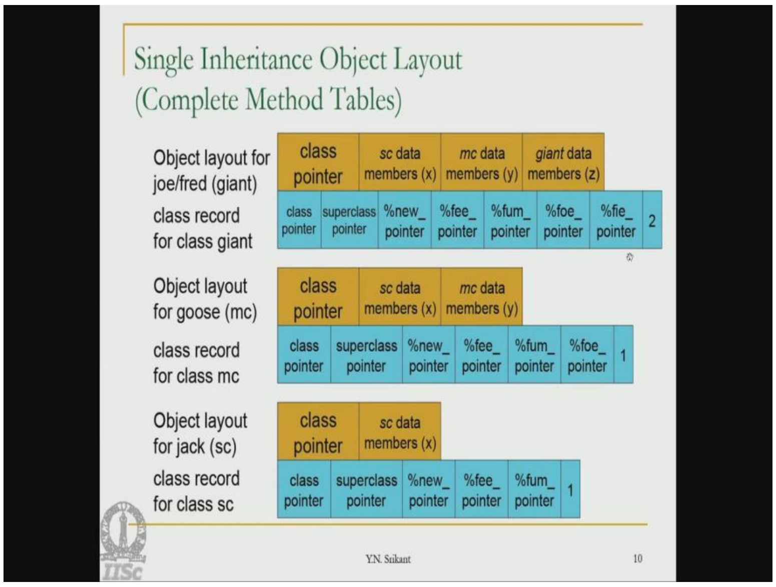
So, let us not consider the method table for the present, let us look at only the yellow part which is the data record. So, there is a class pointer then we have s c data these are the giant you know object data layouts, so then there is s c data, m c data and giant data. So, observe that s c data and m c data is also present in objects of type m c and then only s c data is present only in objects of type s c. But, the most important thing is the space that is provided for the s c data in the you know any object record of type class c must match exactly with the s c data space that is provided in both objects of type m c and objects of type giant, so that is a very important thing.

So, if there are five variables let us say here there is only one if there are five variables in s c and they are listed in a particular order in the s c class record then rather s c object record then they must be listed exactly in the same order in the derived class objects as well that is m c objects and giant objects. The same is true for m c data, when we look at object layouts of you know objects corresponding to class m c and class giant, this of course, will not have any information about m c, because this is the super class and m c is the sub class.

So, the space provided for m c data here and the m c data here must match the amount must match and the offsets must also match. So, the offset of the m c data area is this much, this is the offset it must be the same offset her also and that is why the giant data members actually come at the end, if the giant data members are to be here then you know it would have been really cares. Whenever we call a method corresponding to class s c from an object of class giant it would actually treat the first area as the s c data.

So, if we had giant class data here it would be incorrect data as far as that method is concerned. So, when a class you know the method tables also form follow a similar sequence as above, so and then there is another very important point before we move onto the picture, when a class redefines a method defined in one of it is super classes. The method pointer for that method implementation must be stored at the same offset as the previous implementation of that method in the super class, I am talking about complete method tables, so let us understand what this really means.

(Refer Slide Time: 22:38)

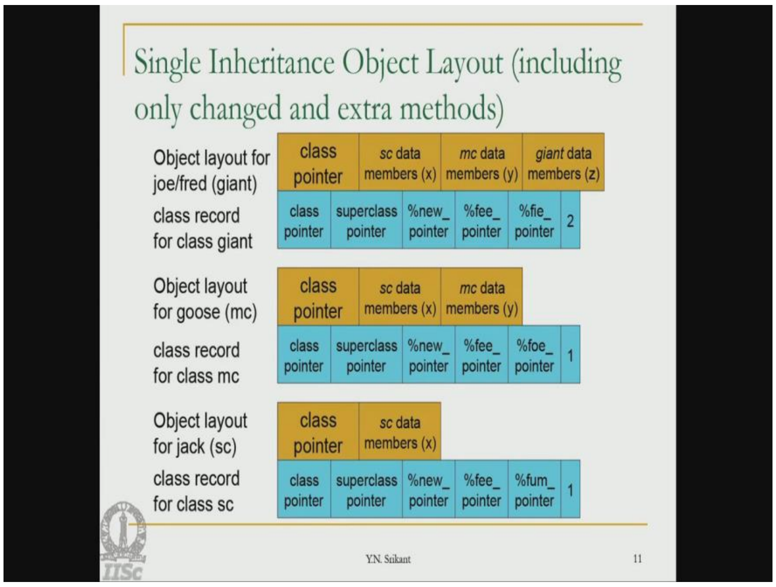


So, here is the method pointer table for the you know inheritance hierarchy here. So, consider the class record for class s c that is the one which stores the method tables, so there is a class pointer of course, then the super class pointer there is a new pointer this would be common for all of these and then the pointer to fee and pointer to fum. Now, what we really say fee of course, redefined by the class m c and which is also redefined by the class giant, but the pointer to fee is now let us say the first, second, third, fourth write in in this class record it must be fourth in the class record for m c and it must be the fourth one in the class record for giant as well.

So, it is, so if it is not then you know it is not possible to statically say this is the correct place at which the pointer to the method fee exists. So, there must be some uniformity in these class records to store the appropriate method pointers, then we have fum pointer here fum pointer is actually implemented by the class s c and it is shared by classes m c and giant as well.

So, the fum pointer is in the next one and it is actually in the same positon as far as class record s c, mc and giant are concerned, so those are the only one's which are provided in class record s c. Now, between m c and giant again we have the pointer foe, so pointer foe must be in the same position as far as relative position as far as these two class records are concerned and then the fie pointer of course, is at the end. So, in a net shell all the information which is shared bit the classes above should be prior to all the information that is provided specially in a particular class. So, fie for example, is a special method provided in the class giant it is not present in the super classes, so that should come actually at the end.

(Refer Slide Time: 25:08)



Single Inheritance Object Layout (including only changed and extra methods)

Object layout for joe/fred (giant)	class pointer	*sc* data members (x)	*mc* data members (y)	*giant* data members (z)		
class record for class giant	class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fie_ pointer	2
Object layout for goose (mc)	class pointer	*sc* data members (x)	*mc* data members (y)			
class record for class mc	class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%foe_ pointer	1
Object layout for jack (sc)	class pointer	*sc* data members (x)				
class record for class sc	class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fum_ pointer	1

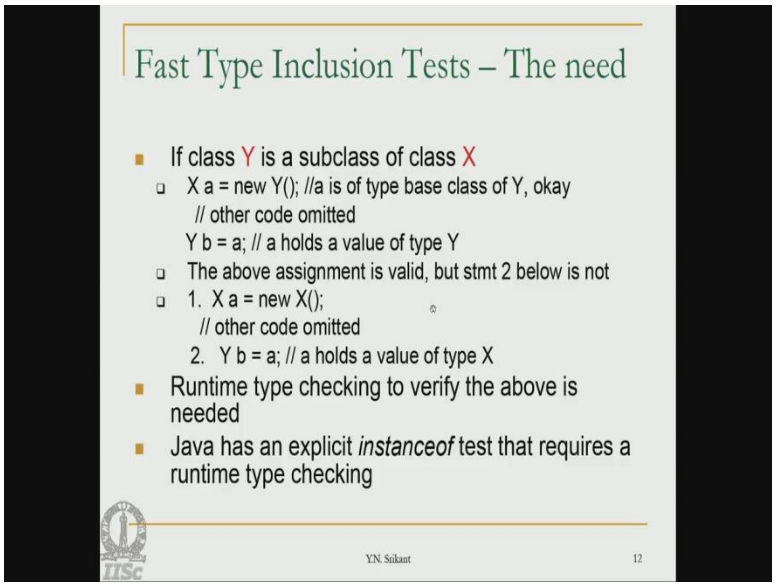
Y.N. Srikant 11

Suppose we do not store the entire method table right, so in that case it really it does not matter. So, we store only the corresponding methods, which are changed or which are new, but whenever we provide for a common method across, so we redefine fee in both m c and giant, so they it must be present in at the same offset as far as all the three class records are concerned. But, then fum is provided only in this, foe only in this and fie only in this ((Refer Time: 25:43)) the rest are all shared.

So, in this type of you know non shared rather incomplete method tables, the class hierarchy is such must be done from here to here. So, whenever there is a method name say example fum it is searched in this table it is not found then it goes to the super class searches in that table it is not found and then it goes to this, it searches in this table found and then it is accessed and executed.

So, you know if we store only the changed and extra methods then the class methods become very compact, but unfortunately the space the time required to access the method increases. Whereas, if we actually store if we store the complete method tables then the space required to store the method tables increases, but the access to methods becomes much, much faster. So, this is the difference between the you know storing the complete method tables and storing only the changed or extra and extra method tables.

(Refer Slide Time: 26:59)



Fast Type Inclusion Tests – The need

- If class Y is a subclass of class X
  - X a = new Y(); //a is of type base class of Y, okay
    // other code omitted
    Y b = a; // a holds a value of type Y
  - The above assignment is valid, but stmt 2 below is not
  - 1. X a = new X();
       // other code omitted
    2. Y b = a; // a holds a value of type X
- Runtime type checking to verify the above is needed
- Java has an explicit *instanceof* test that requires a runtime type checking

Y.N. Srikant 12

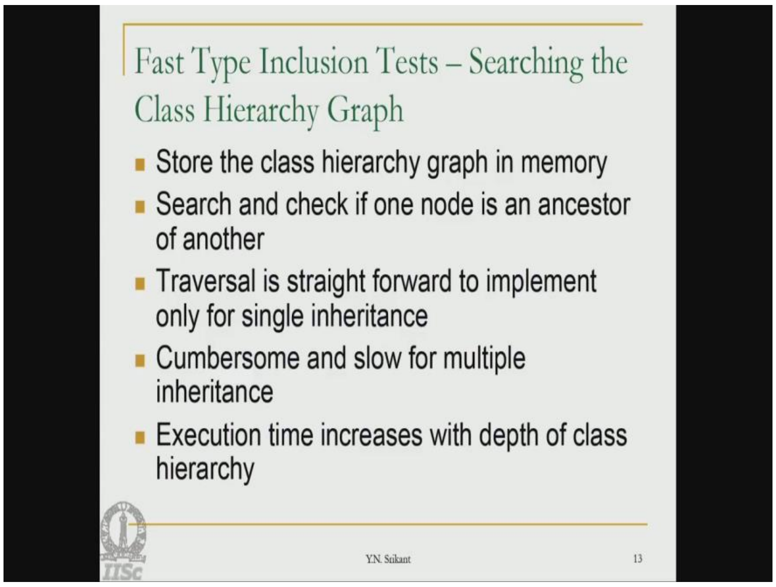
The next important thing that we need to do is to understand how exactly type inclusion tests are put. So, first of all let us understand why type inclusion test is required at all, so the need. So, here is a very simple example, class y is a sub class of class x, so let us assume them now there is a declaration x a equal to new y. So, a is of type base class of y, so either y is the sub class and x is the super class, so x is above y. And now we are actually declaring x a, which is a an object of type super class and we point it to the sub class object new y, so that is what this declaration says.

Then there is a statement y b equal to a, so b is an object of type y that is the sub class and now it points to a, which is again of the same sub class y, so that is not an a issue both are correct. Now, because a even though it is a base class pointer it points to a derived class object which is permitted by the language. And here b is a sub class object derived class object and now it is being assigned a derived class object again which is correct, so this is correct, but the code below here this is wrong.

So, instead of x a equal to new y we said x a equal to new x, so now, a points to a super class object, now we say y b equal to a. So, whereas, b is a sub class object, so we really cannot make it point to a super class object it is always possible to make a super class pointer point to a sub class object, but making a sub class pointer point to a super class object is not permitted, but how does the compiler or the runtime system determine that these objects are of correct types, the pointers are of correct types, this requires runtime type checking.

It is not possible to determine you know the types of b and a in general at compile time in all cases it is necessary to determine these types only at runtime. So, and secondly, java has an explicit instance of test that requires a runtime type checking. So, I can say instance of an given object and then it should return the class to which it belongs to, so that is also something that is very important in the java language and it is programs. So, if for all these reasons we will require methods to do type inclusion test at runtime.

(Refer Slide Time: 30:25)



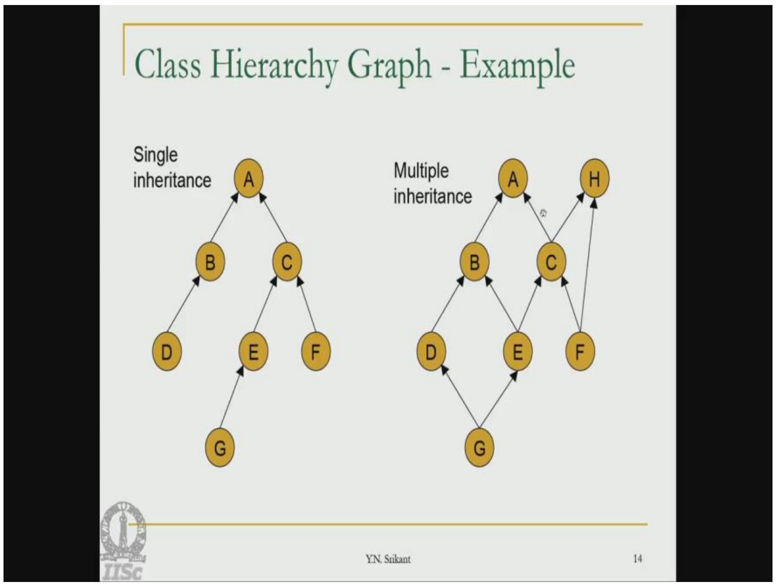
Fast Type Inclusion Tests – Searching the Class Hierarchy Graph

- Store the class hierarchy graph in memory
- Search and check if one node is an ancestor of another
- Traversal is straight forward to implement only for single inheritance
- Cumbersome and slow for multiple inheritance
- Execution time increases with depth of class hierarchy

Y.N. Srikant 13

How does one do this, simplest would be to store the class hierarchy itself, so if you store the class hierarchy graph in memory.

(Refer Slide Time: 30:39)



Class Hierarchy Graph - Example

So, for example, this is a class here hierarchy graph, so this is the highest super class and all these are classes. So, this means single inheritance, whereas this is really multiple inheritance, so if there is single inheritance then you know I have a single parent. So, D has a parent B, B has a parent A. Whereas, if there is multiple inheritance for example, consider E or C right, E has two parents B and C, similarly C has two parents A and H and so on and so forth.

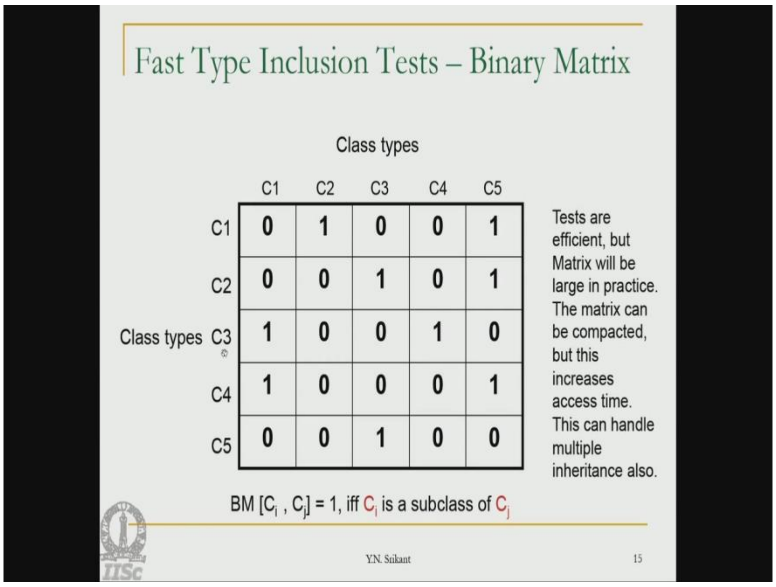
So, multiple inheritance is always very difficult to manage, so and we are not going to study this topic in our course this is meant for advanced study. So, if you have single inheritance how does type inclusion work; suppose I am given an object right then this is a class pointer this node in the class hierarchy graph can be obtained through the class pointer that is not an issue at all.

So, once I get the class pointer I can check its parent and it is parent and so on and so forth. So, now the type inclusion test can be performed if the question is D a sub class of A, we start from D go all the way to route and in the path we get A then the answer is yes, otherwise the answer is no. For example, for D and A we get the answer yes, but for D and E we do not get the answer as yes. So, because we reach A, which is the highest super class, but we did not meet e on the way, so that is why search and check if node is an ancestor of the other.

And the traversal is quite straight forward to implement only for single inheritance, it is very cumbersome and slow for the multiple inheritance, because we need to check all paths execution time increases with depth of class hierarchy. So, for example, if we want to determine the type inclusion for B and C is B an instance of A is C an instance of A etcetera or is B an instance of C whatever the question is if you want to do the inclusion test for nodes at this level then you know one half is enough to determine whether the answer is yes or no.

Whereas, for nodes at this level it requires two halves and for the node at this level it requires three halves to determine the answer. So, if you have deep hierarchy than the amount of time needed to determine type inclusion would be more, so for the nodes which are at the lower level.

(Refer Slide Time: 33:38)



Fast Type Inclusion Tests – Binary Matrix

Class types

Class types	C1	C2	C3	C4	C5
C1	0	1	0	0	1
C2	0	0	1	0	1
C3	1	0	0	1	0
C4	1	0	0	0	1
C5	0	0	1	0	0

Tests are efficient, but Matrix will be large in practice. The matrix can be compacted, but this increases access time. This can handle multiple inheritance also.

BM [C_i , C_j] = 1, iff C_i is a subclass of C_j

Y.N. Srikant 15

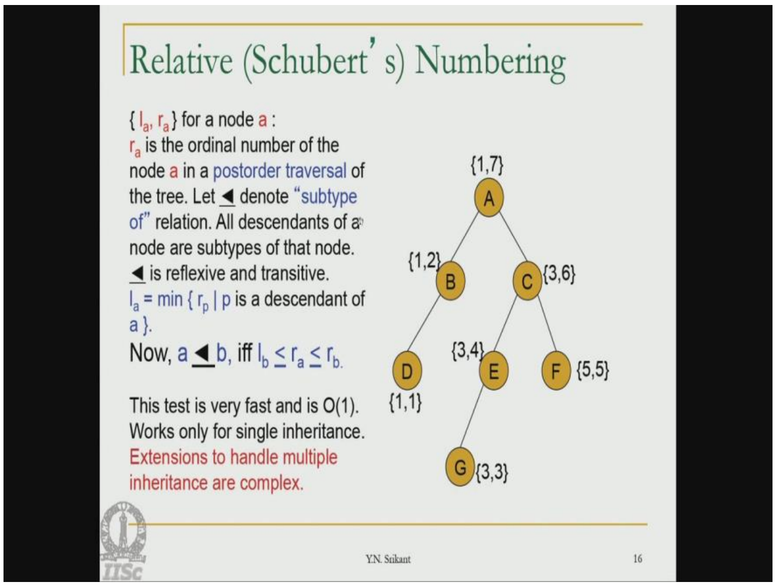
Another way of a storing information about the class hierarchy is to use a binary matrix, so the matrix has columns which are named as the after the classes again it has rows which are again named after the classes. So, class types on this side and class types on this side as well and if we have say class C 3 and we want to check whether it is a descendant of class C 4 we just have to look up the entry B M of C 3 comma C 4. So, if the entry says 1, then yes indeed the answer is yes and if the entry is 0 then the answer is no. So, BM of C comma C j equal to one if and only if C i is a sub class of C j.

So, even the transitivity is taken care of here. So, it is just not one level, but all the levels of the hierarchy are coded as 1 or 0 inside this binary matrix. Tests are very efficient, but the matrix will be very large in practice right, so we have this matrix which is quite large and the space required for the matrix, once it is very large, thus access method you know actually has to be cut down will be quite low, but to store very large matrices we actually use some compaction. So, if the matrix is compacted then you know BM accessing B M of C i comma C j is not going to be an one time access, but it will be much more than that.

So, if the matrix is available in the form that is shown here, then accessing it is very easy, but if it is coded then you know access time also increases of course, this major advantage of this matrix approach is that it can handle multiple inheritance as well.

Whereas, searching the class hierarchy using the tree or the graph was much more difficult.

(Refer Slide Time: 36:03)



Relative (Schubert's) Numbering

{ l_a, r_a } for a node a :
r_a is the ordinal number of the node a in a postorder traversal of the tree. Let $\underline{\blacktriangleleft}$ denote "subtype of" relation. All descendants of a node are subtypes of that node.
$\underline{\blacktriangleleft}$ is reflexive and transitive.
$l_a = \min\{ r_p \mid p \text{ is a descendant of } a \}$.
Now, a $\underline{\blacktriangleleft}$ b, iff $l_b \le r_a \le r_b$.

This test is very fast and is O(1).
Works only for single inheritance.
Extensions to handle multiple inheritance are complex.

Y.N. Srikant 16

There is also another very elegant method which works only for single inheritance and extensions to handle multiple inheritance are very difficult very complicated this is called relative or Schubert's numbering. So, basically we are given the class hierarchy that set compile time., so we do not have to store the class hierarchy at runtime if we have to then the method has no advantage. So, let us assume that there is a single inheritance and the class hierarchy tree is given to us, so here is the class hierarchy tree.

Now, we do a post order numbering of the nodes of the class hierarchy tree, so the second component in each one of these pairs is the post order numbering so; obviously, post order walk over this tree goes to starts with D. So, this gets 1, then it goes to parent it gets 2, then it goes to a parent which has a right child. So, we go down the right sub tree all the way to g that gets a number 3, then the this E gets number 4 and then we go to C and go down the right sub tree that gets number 5, then C gets number 6 and A gets number 7, so this is the first step as far as you know this relative numbering goes.

Now, we need to determine the first component, so second component is already determined this is the ordinal number of the node a in post order traversal of the tree. So, what is the first component l a. So, to define that we must define this relation called sub type of, so let this angle you know this bracket with an underscore denote the subtype of relation. So, all the descendants of a node are subtypes of that node so; obviously, that is true for a inheritance tree anyway.

So, this subtype of relation; obviously, is reflexive and transitive that is very easy to see you know. So, we have D is a subtype of B and B is a subtype of A, so D is; obviously, subtype of A and every node is a subtype of itself. So, reflexive and transitive properties are true, what is l a? L a is minimum of r p such that p is a descendant of a. So, if I consider a node such as a, then we look at the post order numberings of each of the node find out the minimum.

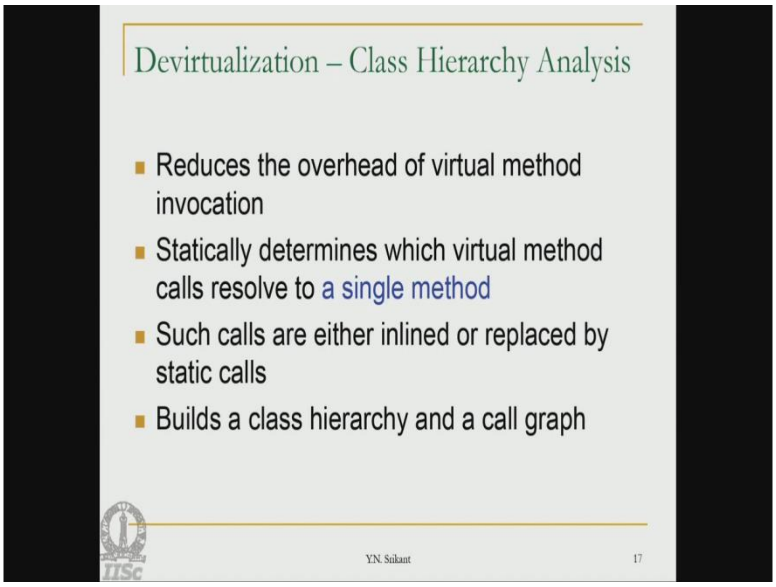
So, in this case; obviously, 1 is the minimum, so one happens to be l a of this particular node the same is true for b it is among it is descendants one is the minimum. So, one is a is the you know the first component of this as well; obviously, one is a first component first component of D as well. When we come to c the minimum happens to be 3 again. So, three is the first component of this node, this node and this node and for this node this I this is born it is own, so 5 is the only thing appears, so 5 is the first component of this node.

Now, the test says a is subtype of b if and only if r a is between l b and r b. So, if I take the post order numbering of the node A, it must be located between l b and r b of node B that is what it says, so let us check it out let us check it out ford and b. So, the post order numbering of d is 1 this is between 1 and 2 perfectly correct. So, D is subtype of B and; obviously, D is a subtype of A, because one you know A is between 1 and 7. Now, let us consider B and E, so E a subtype of B, the you know post order numbering of E is 4 it is; obviously, not between not 1 and 2.

So, E is definitely not a subtype of B and E is B a subtype of E that is also not true because the post order numbering of B is 2 and it is definitely not between 3 and 4, so that is also false. So, this is the very simple check, once we have at a compiled time, once we have the inheritance tree, we do the post order walk compute the second component, then you know we compute the minimum and compute the first component. So, now, the entire tree is decorated we can store the numbers this Schubert’s numbers or relative numbers of each one the nodes along with their class pointers that is very easy.

So, given the class pointer you know we access the relative number and then determine whether this relation l you know rather this inequality holds l b less than r a less than or equal to r p. So, this is an order one computation very fast, but it is and of course, remember that this is if and only if. So, if this holds then subtype holds and if this is a subtype, then this inequality holds, this holds only for works only for single inheritance, but that is what we are really studying today, so that is you know that that is about type inclusion.

(Refer Slide Time: 42:16)



Devirtualization – Class Hierarchy Analysis

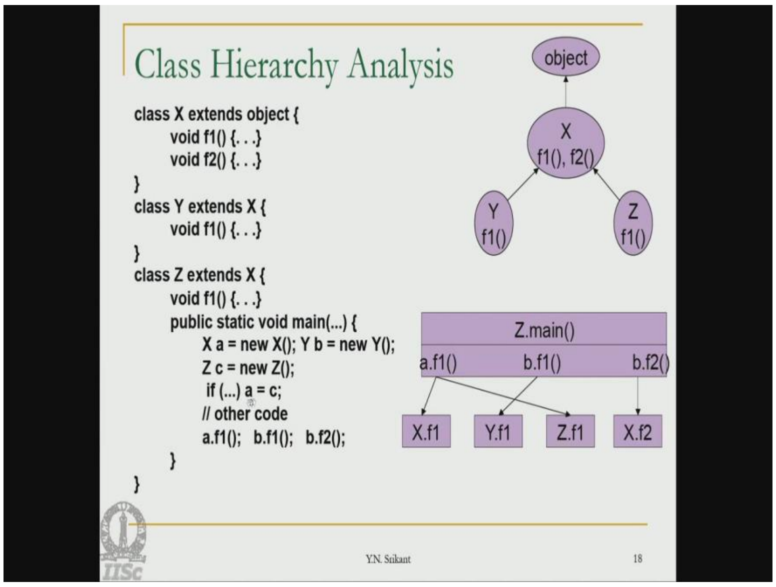
- Reduces the overhead of virtual method invocation
- Statically determines which virtual method calls resolve to a single method
- Such calls are either inlined or replaced by static calls
- Builds a class hierarchy and a call graph

Y.N. Srikant 17

Let us look at one optimization which is performed on object oriented languages, which is a fairly straight forward, but very important optimization this is called devirtualization and it can be performed using what is known as class hierarchy analysis. Basically, it reduces the overhead of virtual method invocation and statically determines which virtual method calls can resolve to a single method, so such calls are either inline or replaced by static calls.

So, basically we want to avoid searching the method tables at runtime, but this can be done only in certain numbers of cases not all cases, whenever we can do it we know the target, we replace the method table search a you know, which can be avoided and we can replace that by static call an ordinary procedure call of course, this requires a class hierarchy and a call graph.

(Refer Slide Time: 43:36)



Class Hierarchy Analysis

```
class X extends object {
      void f1() {. . .}
      void f2() {. . .}
}
class Y extends X {
      void f1() {. . .}
}
class Z extends X {
      void f1() {. . .}
      public static void main(...) {
           X a = new X(); Y b = new Y();
           Z c = new Z();
            if (...) a = c;
           // other code
           a.f1();  b.f1();  b.f2();
      }
}
```

Y.N. Srikant 18

Let me show you an example of how to do that, here is a small program here is class X that extends the object in java. So, here is object, then X is derived from it has two methods f 1 and f 2 then y is derived from x and it has a it redefines a method f 1, then again Z you know inherits from X and it redefines the method f 1 as well. Now, within the class Z, there is a main program, so that is the Z dot main Z dot main has three calls a dot f 1 b dot f 1 and b dot f 2. So here, so for example, X a equal to new X Y b equal to new Y and Z c equal to new Z.

So, if something than a equal to c right, so a is actually an object of type class X and c is actually an object of type class Z, now we say a equal to c. So, now, a can point to either type Z node or Z object or it can point to you know a type x object either one of them, so that is the difference. So, if we do not know to which one of these it points to we may not be able to determine statically the method that is actually executed from a and that is precisely what happens here, so we call a dot f 1.

So, when we do this if it was pointing to an object of type x then it would be x dot f 1, but if that was pointing to an object of type z then it would have been z dot f 1. So, the call graph points to you know a dot f 1 points to both x dot f 1 and z dot f 1 and there is no way you can actually resolve this call at compile time. But, you should take b dot f one it always points to only y dot f 1 rather it call the call is always y dot f 1 not points to. So, there is no change in what b points to similarly b dot f 2 is always x dot f 2, because b does know this class y and class z do not define their own f 2 it is always in x.

So, b dot f 1 can be you know the there is no need to search any method tables here it can be directly coded as y dot f 1 and b dot f 2 can be also coded directly as x dot f 2. So, these become static calls. Whereas, a dot f one cannot be resolved at compiled time one has to generate code to check the type of a if the type of a is X, then the call material realizes as x dot f 1 and if the type of a is Z then the call would be translated to Z dot f one. So, this runtime resolution will be definitely necessary in certain cases not in all cases and whenever we can resolve the call to a static call we can do it all, so that is about our you know object oriented language implementation.

(Refer Slide Time: 47:21)

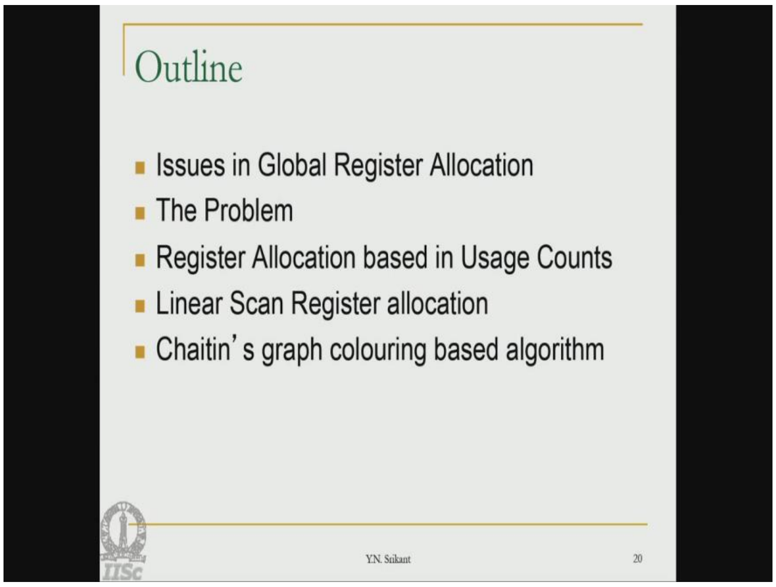
Global Register Allocation - 1

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Principles of Compiler Design

Now, let us move on to a new topic Global Register Allocation.

(Refer Slide Time: 47:27)



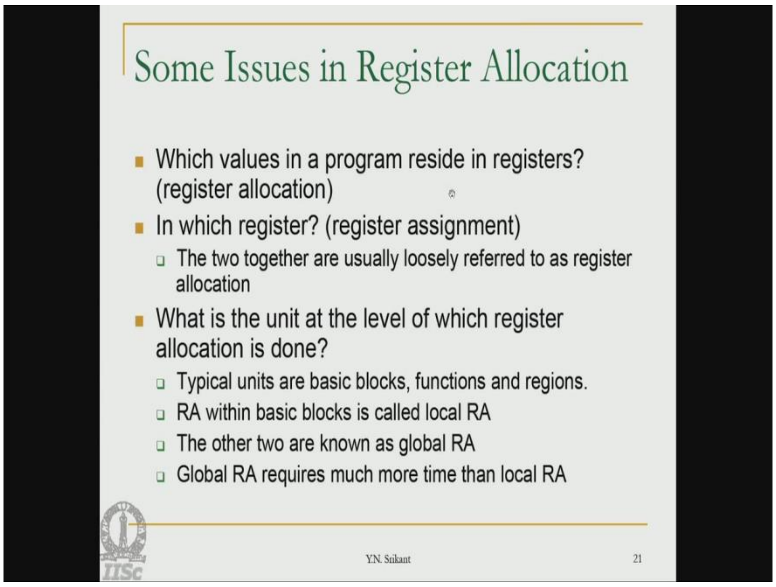
Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin's graph colouring based algorithm

Y.N. Srikant 20

So, in global register allocation, we must understand why we require global register allocation what are the various issues define the problem properly. And then we study three methods of doing global register allocation the and it is in the increasing order of complexity. First one is based on what are known as usage counts, the second one is known as a linear scan register allocator and third is based on graph coloring.

(Refer Slide Time: 48:04)



Some Issues in Register Allocation

- Which values in a program reside in registers? (register allocation)
- In which register? (register assignment)
  - The two together are usually loosely referred to as register allocation
- What is the unit at the level of which register allocation is done?
  - Typical units are basic blocks, functions and regions.
  - RA within basic blocks is called local RA
  - The other two are known as global RA
  - Global RA requires much more time than local RA

Y.N. Srikant 21

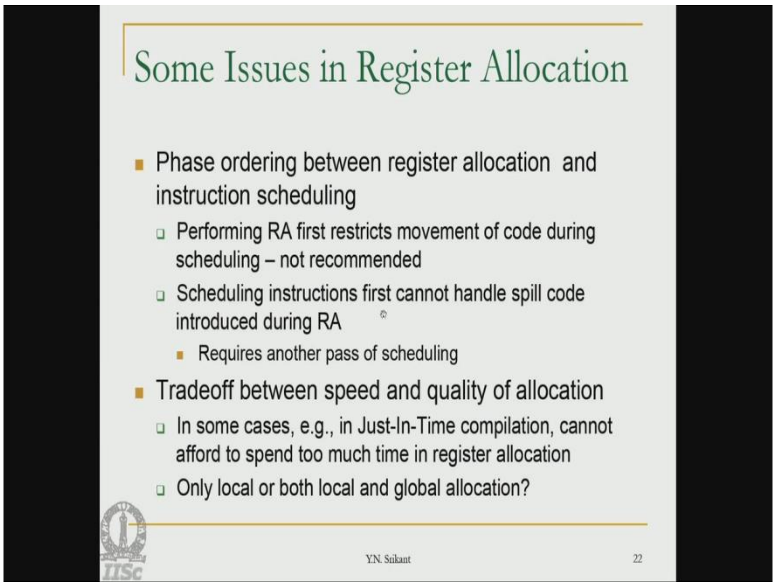
Let us understand the issues in register allocation, the question to be asked is which values in a program reside in a registers. So, now we are looking at a register allocation as such, so there may be ten variables ten values which are used in the program, when the program runs which of these values must reside in registers, otherwise obviously, it would reside in memory.

So, to make an intelligent decision about this the of course, the best answer would be take those variables which are accessed you know very frequently and maximum number of times. And then make sure that the you know the runtime of the program the execution time of the program is minimized when we place these values in registers that is the correct way of doing it, but can we actually do all this assessment at compiled time and make the allocation. So, that the execution time is minimized not necessarily, the reason is the allocation of registers is very complicated task, we can only do it an approximate way the problem in general is m p complete.

Then the second question to be answered is in which register, so this called as the register assignment problem. So, usually the two are loosely referred to as register allocation, because in modern architectures the except for one or to two registers all other registers can be used interchangeably, if there are very special purpose registers then this problem becomes more difficult. What is the unit at which you know register allocation is done; for example, if it is a local register allocator then it does register allocation at the level of basic blocks. Whereas, if it is a global register allocator then it does it at the level of a function or it does it at the level of group of functions called regions or of course, it could also be group of basic blocks in a program.

And that would be a loop or a nested loop or something like that register allocation within basic blocks is called local register allocation and the other two is called as global register allocations. Now, global register allocation obviously, requires much more time than local register allocation and all register allocations are actually carried out at compiled time of course, there are you know exceptions to this we will see that very soon.

(Refer Slide Time: 51:02)



Some Issues in Register Allocation

- Phase ordering between register allocation and instruction scheduling
  - Performing RA first restricts movement of code during scheduling – not recommended
  - Scheduling instructions first cannot handle spill code introduced during RA
    - Requires another pass of scheduling
- Tradeoff between speed and quality of allocation
  - In some cases, e.g., in Just-In-Time compilation, cannot afford to spend too much time in register allocation
  - Only local or both local and global allocation?

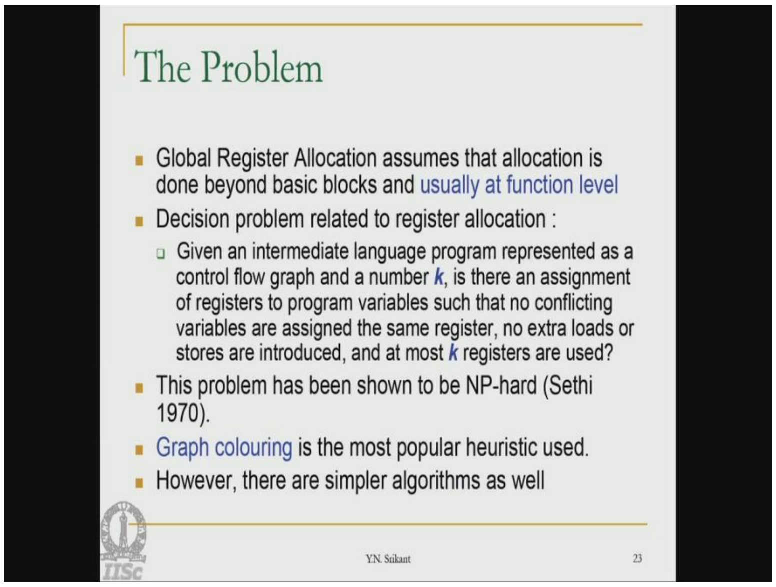
Y.N. Srikant 22

Now, phase ordering between register allocation and instruction scheduling is a well know problem, instruction scheduling is nothing but reordering of instructions, so that pipeline hazards are eliminated. So, if we do register allocation first then the movement of code during instruction scheduling is restricted, so this is not know you know recommended. Whereas, if we you know scheduled instructions first then during register allocation, we may have to introduce extra code called spill code and that would not be scheduled at all.

So, what we do as practical in practical situations is do scheduling first then do register allocation and then require do scheduling again, so to handle this spill code, so two passes of scheduling would be done. Now, there is a tradeoff between speed and quality of allocation as well even at whether it is compiled time or whether it is runtime, so let us see what happens. So, in some cases for example, in the case of just in time compilation of java, compiled time is actually you know limited to producing the java white code, but during the execution of the java white code, there is a phase of machine code generation which is called just in time compilation which is performed to increase the speed of that code.

So, in such a case if we perform a very complicated register allocation task within the just in time compiler, it would really require too much time you know. So, performing very complicated register allocation in just in time compilation you cannot be justified in such cases that is the first issue. Second issue is should be performed both local and global allocation, yes that is possible we could actually do global allocation first and then perform local allocation.

(Refer Slide Time: 53:22)



The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level
- Decision problem related to register allocation :
  - Given an intermediate language program represented as a control flow graph and a number *k*, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most *k* registers are used?
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well

Y.N. Srikant 23

Now, let us define the problem of register allocation and see what it really means. So, global register allocation assumes that allocation is done beyond basic blocks and usually at the function level and the problem is you are given an intermediate language program it is represented as a control flow graph you are given you know k registers as well. So, that is the number of registers available, is there an assignment of registers to program variables such that no conflicting variables are assigned in same register.

In other words two variables which actually alive at the same time should not be given the same register of course, there must be no extra loads or stores introduced because of this assignment and at most k registers are used you could use less than k registers, but you cannot definitely use more than k registers. So, this problem, so that is again let me summarize you are given k registers there are many variables in the program. So, what can we is there an assignment of program variables such that no conflicting variables are assigned the same register and we do not use more than k registers at that time at any point in time.

So, this problem has been shown to be n p hard by Ravi Sethi and we use a very popular heuristic called graph coloring to solve this problem, but there are other many other simpler algorithms as well. So, the difficulty here is if there are very large number of registers, then you know either it is possible to assign variables to registers very easily. The problem start when there are very few registers and we have to intelligently assign variables to the registers. So, some of the registers some of the variables cannot be assigned registers, so they will have to remain in memory, so that is the basic problem here.

Graph coloring is used as a popular heuristic to perform this allocation. So, in graph coloring to a very briefly the colors are the registers, we construct what is known as a conflict graph and then apply reduction on the conflict graph to achieve coloring. So, we will stop at this point and continue with the discussion on graph coloring in the next lecture.

Thank you.