Implementing Object-Oriented Languages - 2

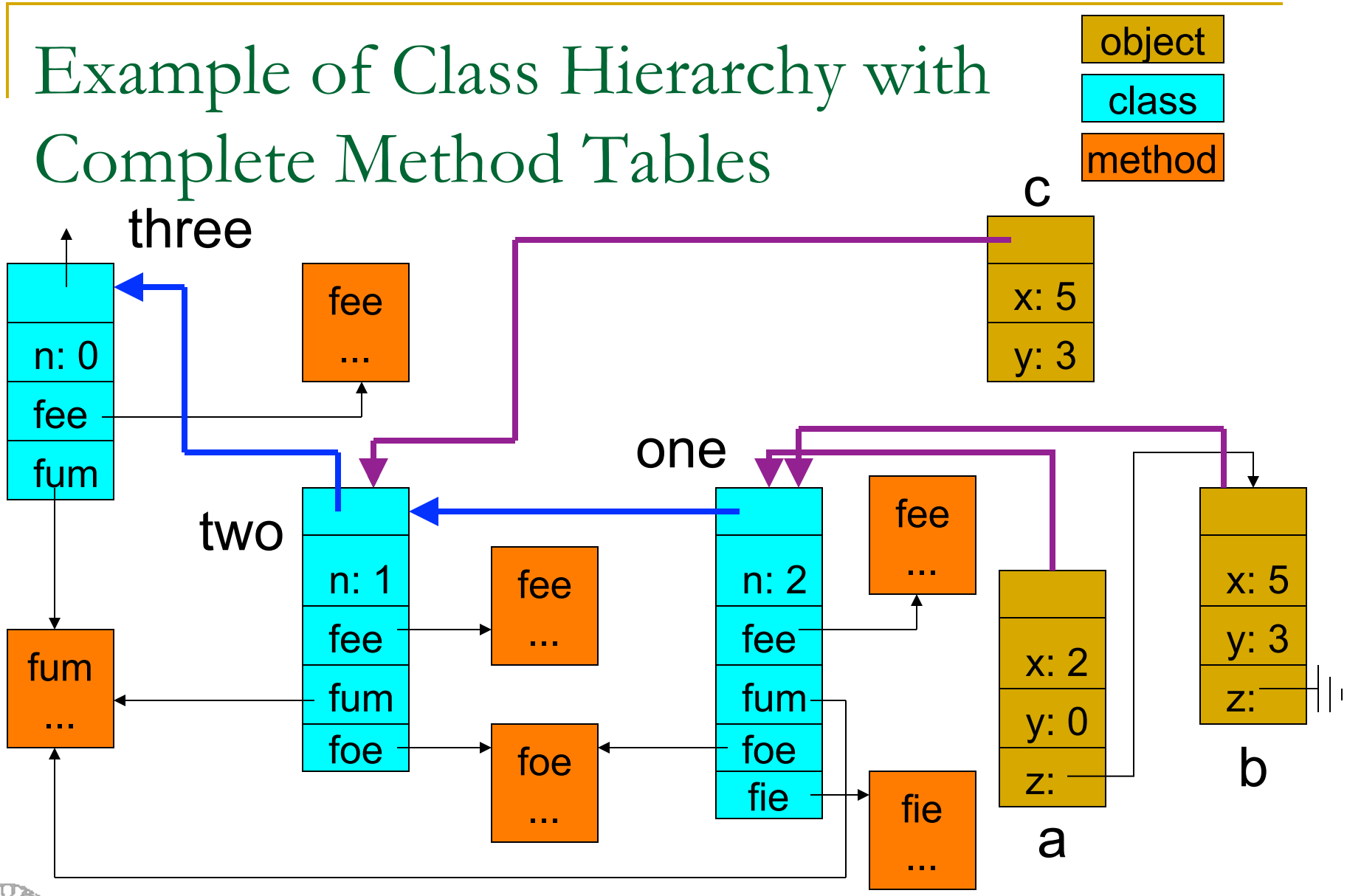
Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

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.**

Example of Class Hierarchy with Complete Method Tables



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)

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

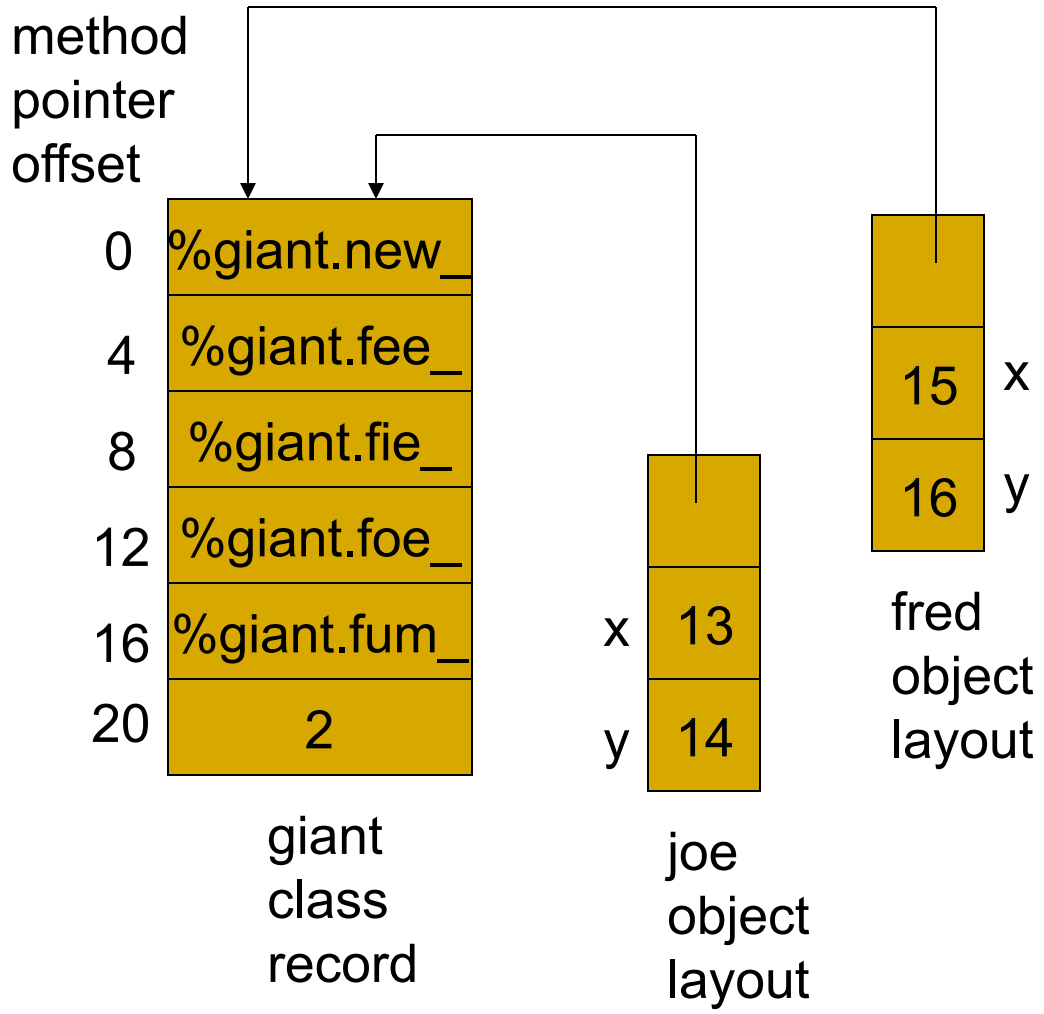
Code Generation for Methods

- 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

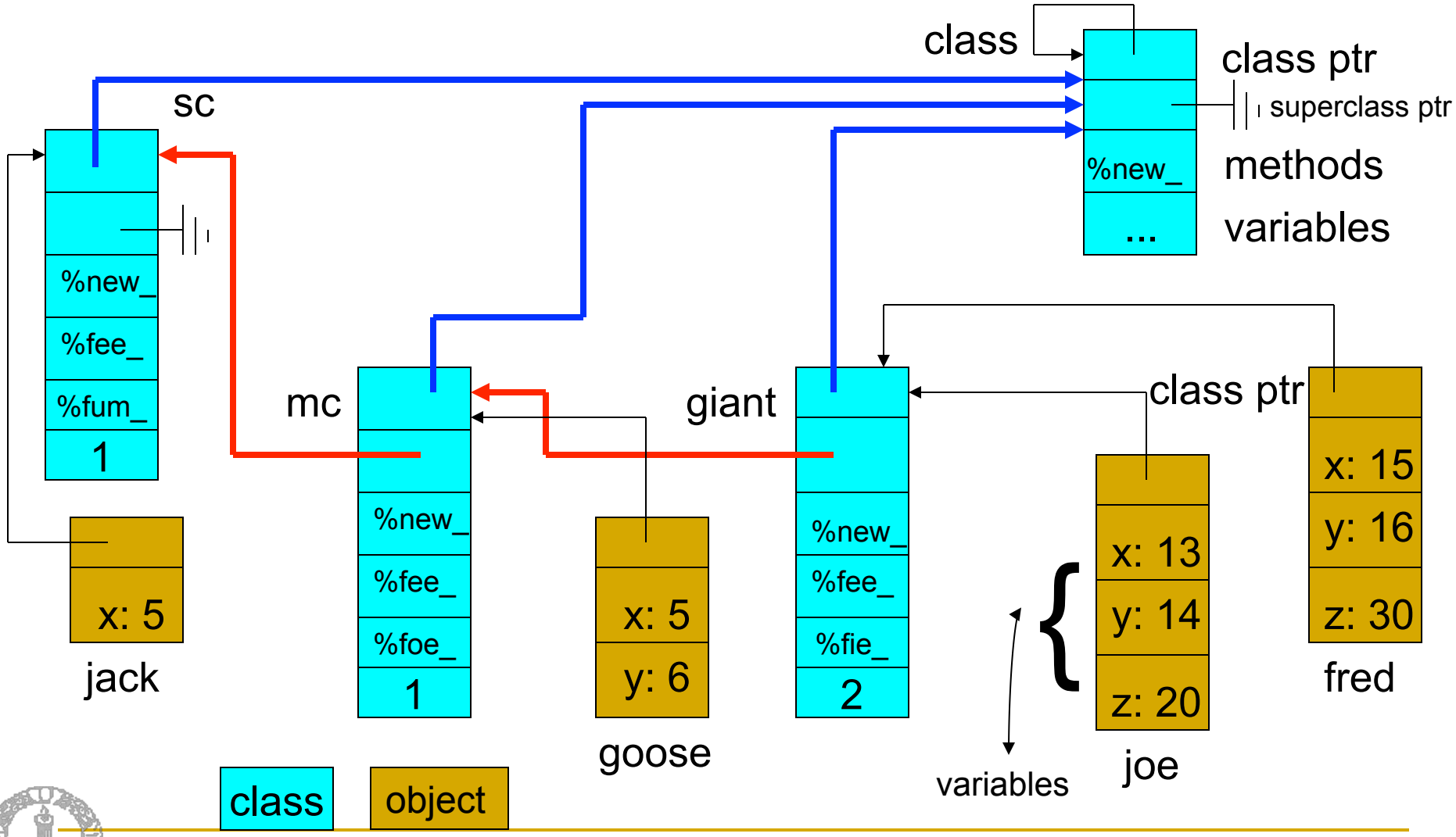
Single Class, No Inheritance

Example:

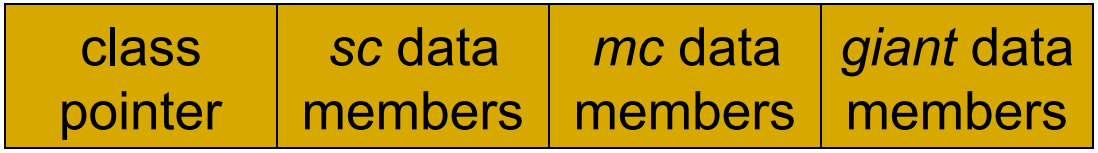
```
Class giant {
    int fee() {...}
    int fie() {...}
    int foe() {...}
    int fum() {...}
static n;
int x,y;
}
```



Implementing Single Inheritance



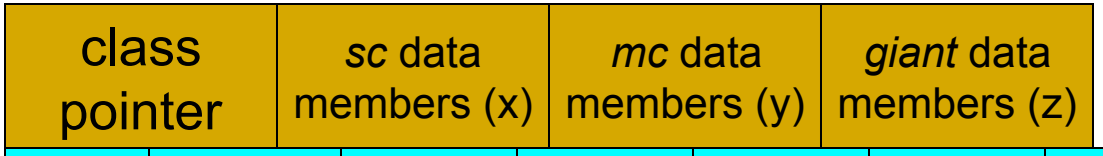
Single Inheritance Object Layout



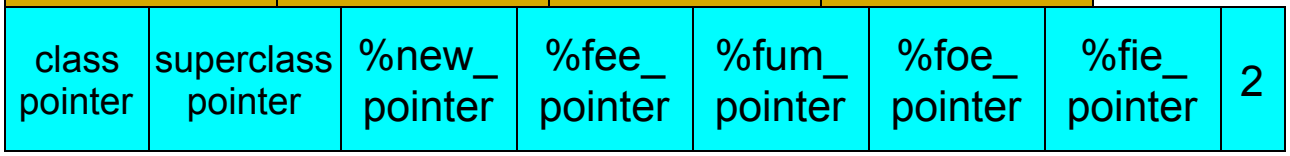
- Now, an instance variable has the same offset in every class where it exists up in its superclass
- Method tables also follow a similar sequence as above
- When a class redefines a method defined in one of its superclasses
  - the method pointer for that method implementation must be stored at the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

Single Inheritance Object Layout (Complete Method Tables)

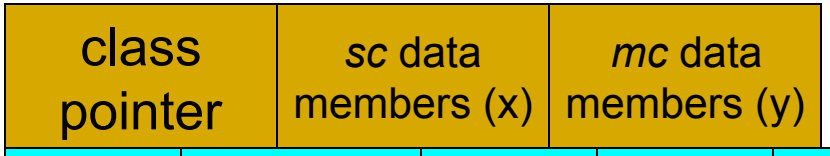
Object layout for joe/fred (giant)



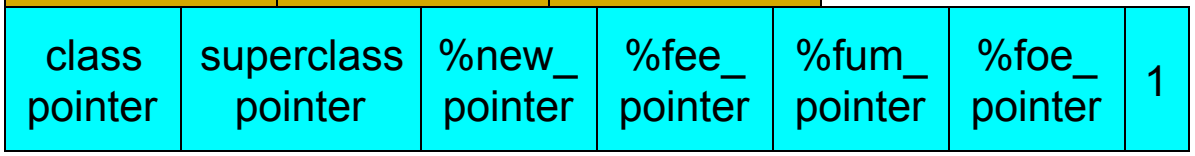
class record for class giant



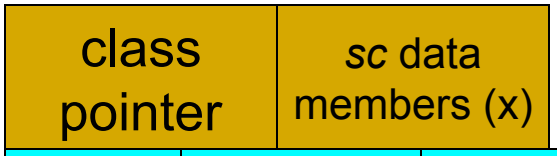
Object layout for goose (mc)



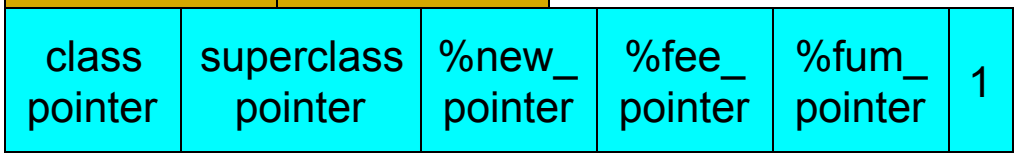
class record for class mc



Object layout for jack (sc)

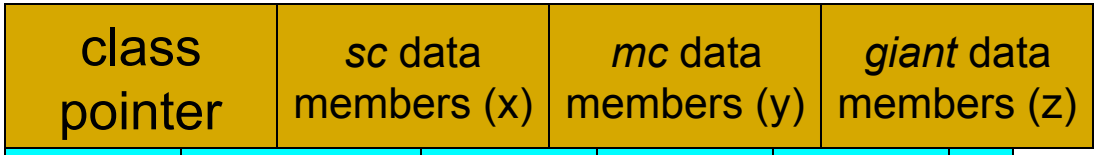


class record for class sc

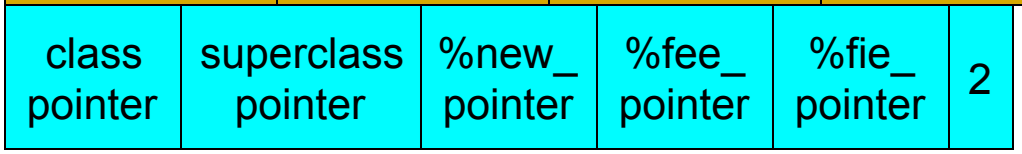


Single Inheritance Object Layout (including only changed and extra methods)

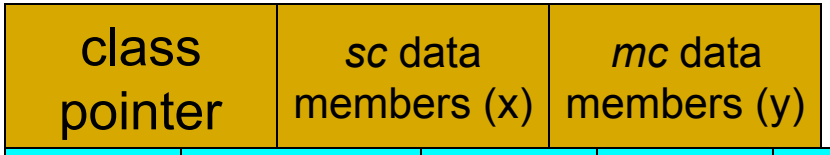
Object layout for joe/fred (giant)



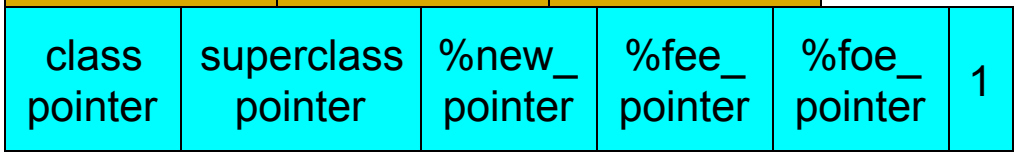
class record for class giant



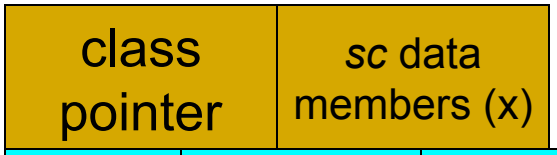
Object layout for goose (mc)



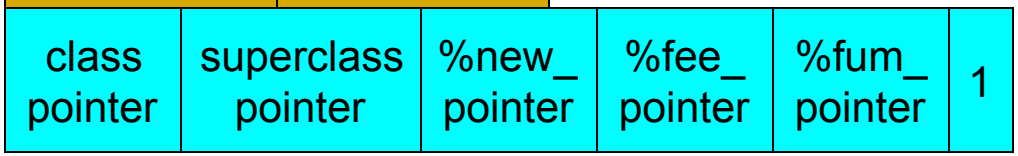
class record for class mc



Object layout for jack (sc)



class record for class sc



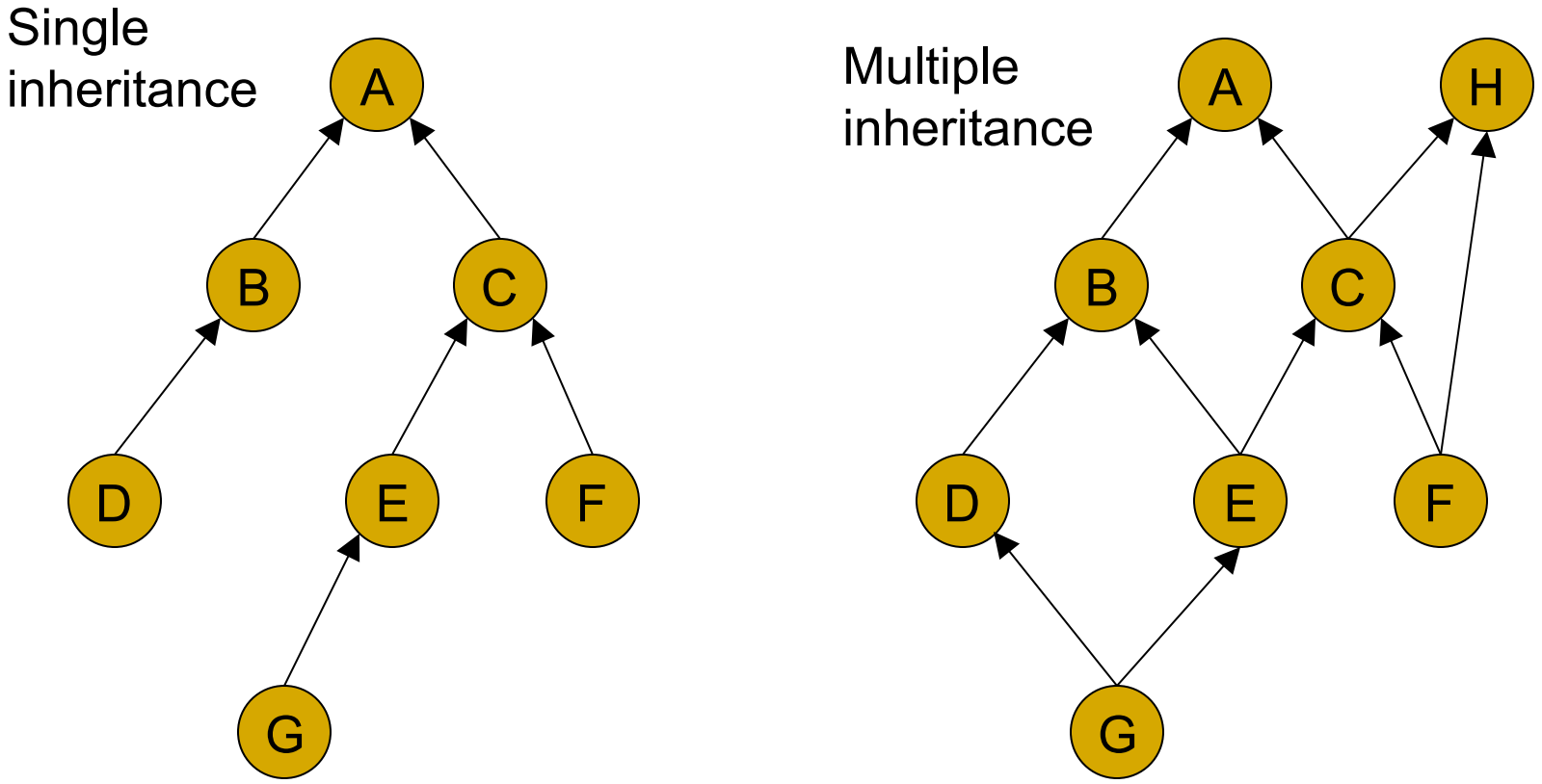
Fast Type Inclusion Tests – The need

- 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

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

Class Hierarchy Graph - Example



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**

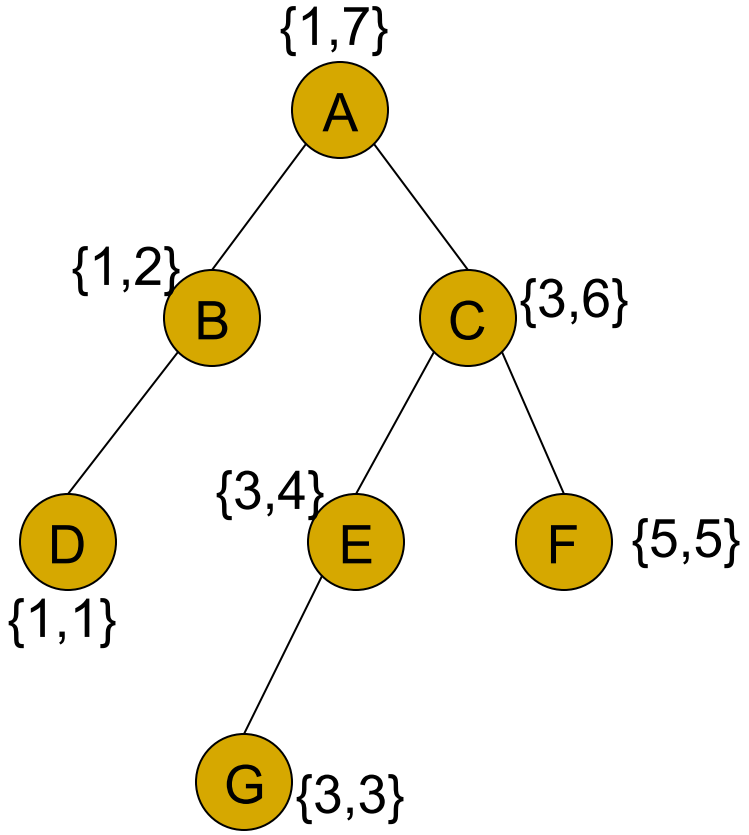
Tests are efficient, but Matrix will be large in practice. The matrix can be compacted, but this increases access time. This can handle multiple inheritance also.

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

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 $\preceq$ denote "subtype of" relation. All descendants of a node are subtypes of that node. $\preceq$ is reflexive and transitive.
l_a = min { r_p | p is a descendant of a }.
Now, a $\preceq$ b, iff $l_b \leq r_a \leq r_b$.

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

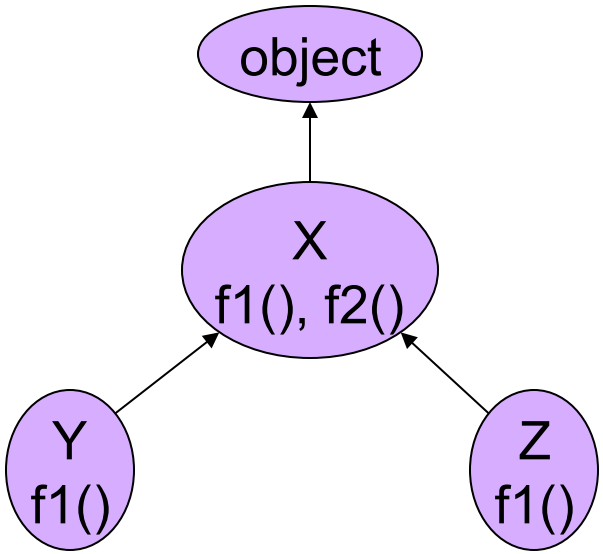


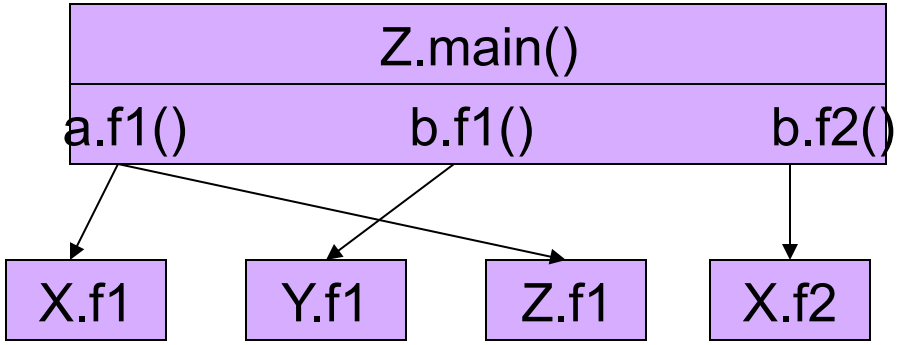
Devirtualization – Class Hierarchy Analysis

- 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

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();
        }
}
```





Global Register Allocation - 1

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

NPTEL Course on Principles of Compiler Design

Outline

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

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

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?

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