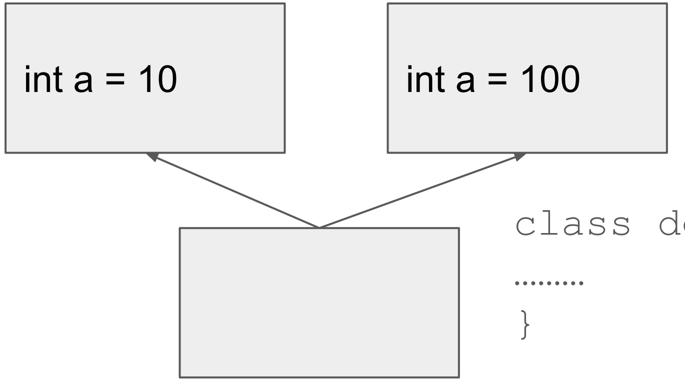
CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

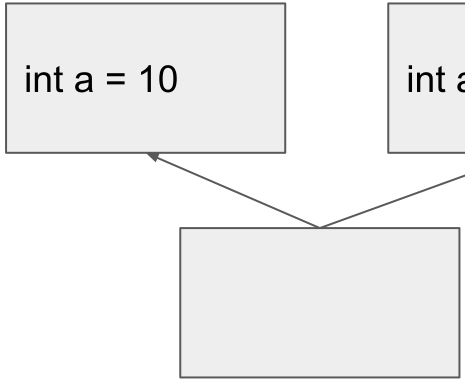
Problem of Inheriting More Than One Base Class (C++)



```
class derived: public base1, public base2{
………
}
…
derived ob;
cout << ob.a;
```

Which version of a is being referred to?

Problem of Inheriting More Than One Base Class (C++)



```
class derived: public base1, public base2{
………
}
…
derived ob;
cout << ob.a;
```

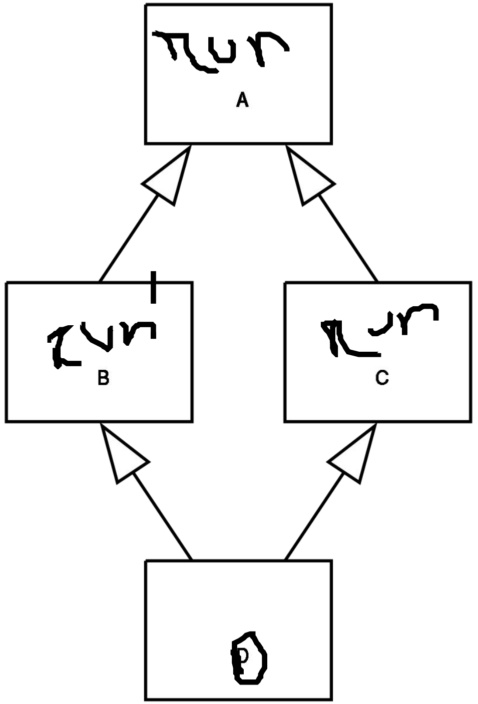
Which version of a is being referred to?
Answer: ambiguous, compile time error. But if you don't write the statement `ob.a`, then no error.

The “Diamond” Problem

As an example, in the context of GUI software development, a class Button may inherit from both classes Rectangle (for appearance) and Clickable (for functionality/input handling), and classes Rectangle and Clickable both inherit from the Object class. Now if the `equals()` method is called for a Button object and there is no such method in the Button class but there is an overridden equals method in Rectangle or Clickable (or both), which method should be eventually called?

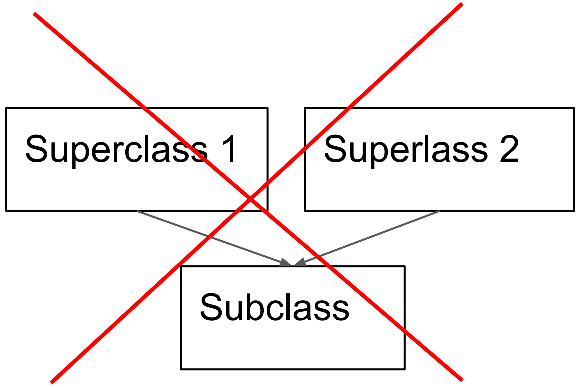
It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation. In this case, class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape.

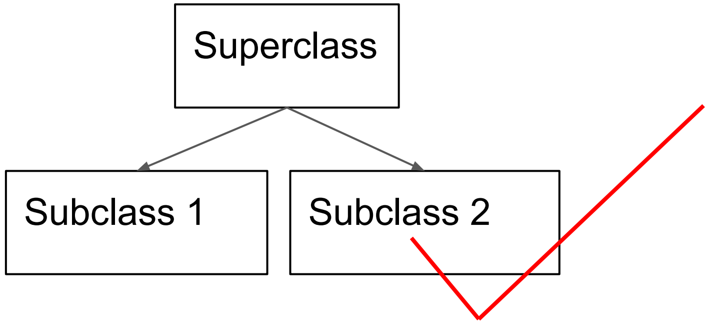
https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

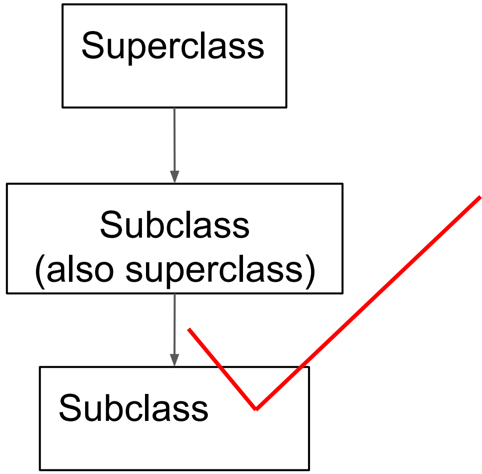


Multiple Inheritance

- Unlike C++, Java doesn’t support a subclass to inherit more than one superclass.
  - Why? Because its drawbacks outweigh its benefits (Java developers thought).
    - What to do then if a class conceptually inherit properties of more than one class? Ans.: Not too common in real life. (1) Change the object hierarchy (2) Interface alleviates the problem to some extent (to be discussed later).
- However, a single superclass can be inherited by more than one class.
- Multi-level inheritance is also allowed, like C++.





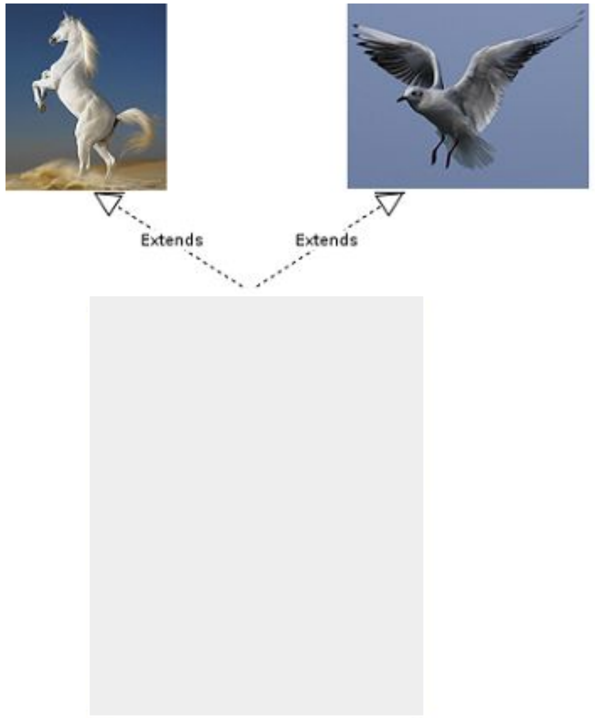


A Historical Note

The paper entitled “Java: an Overview” by James Gosling in February 1995 gives an idea on why multiple inheritance is not supported in Java:

“JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.”

https://javapapers.com/core-java/why-multiple-inheritance-is-not-supported-in-java/

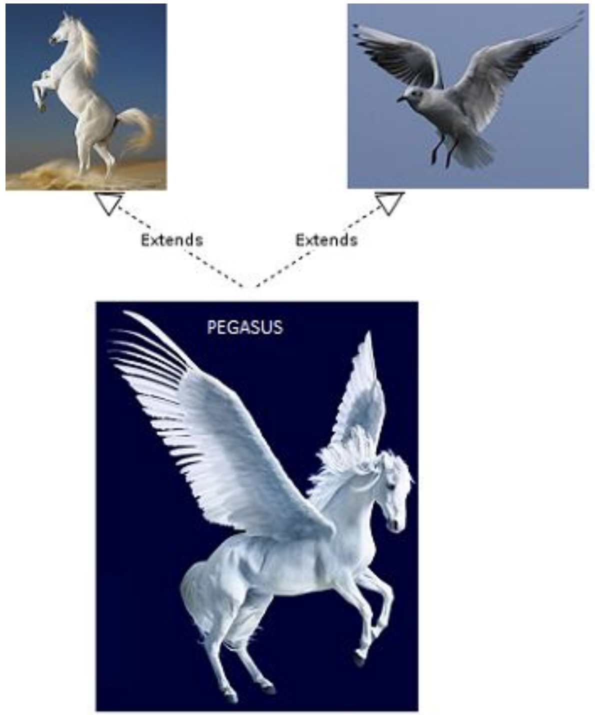


A Historical Note

The paper entitled “Java: an Overview” by James Gosling in February 1995 gives an idea on why multiple inheritance is not supported in Java:

“JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.”

https://javapapers.com/core-java/why-multiple-inheritance-is-not-supported-in-java/

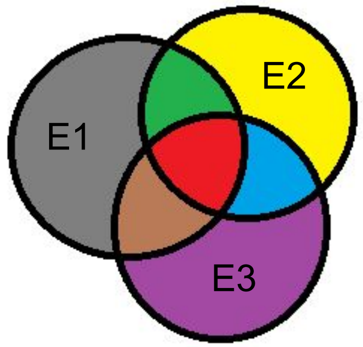


How to Compartmentalize Abstract Concepts into Classes?

Name	Roll	Address	SSC Marks	HSC Marks	No. of Siblings	Age	Father’s Name	Mother’s Name	Gender	Bank Account No.	Ph. No.	… …

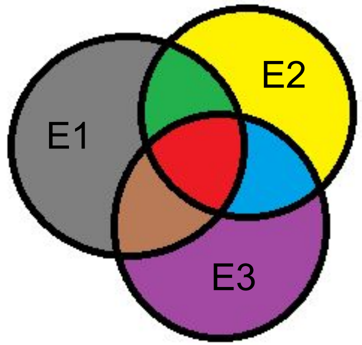
One big table or multiple smaller tables? Which categorization to choose?

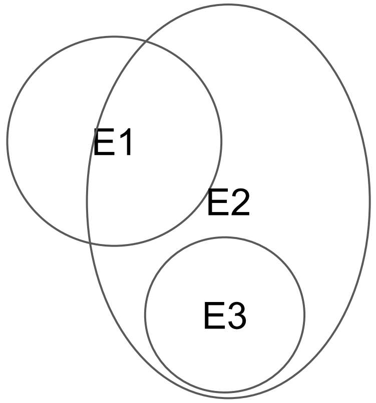
How to Compartmentalize Abstract Concepts into Classes?

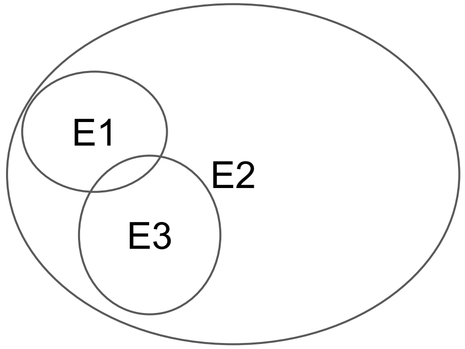


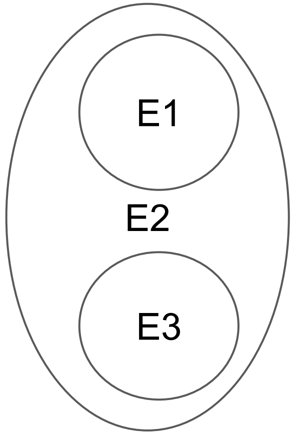
Consider three imaginary entities: E1, E2 and E3. Ideally, they may have some exclusive properties, some properties shared by any two of them, and some other properties shared by all of them.

How to Compartmentalize Abstract Concepts into Classes?



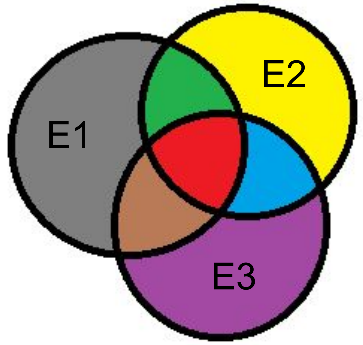






Different scenarios of "sharedness" of properties among entities.

How to Compartmentalize Abstract Concepts into Classes?



One entity sharing from two unrelated entities -- too ideal situation! (Java designers thought).

- Not very common in programming.
- Rather creates problem if two immediate superclasses have the same function: the so-called “diamond” problem.
- So if it does occur, slightly modify your object hierarchy to avoid multi-class inheritance.
  - Maybe at the cost of reduced code-reuse.
  - Remember, however, that code-reuse is aimed at facilitating programmers, so if additional problems are created, then benefits and limitations must be weighed.

Note: Many of the students fail (and more importantly, does not feel the importance!) to apply a proper breakdown of concepts into classes.

How to Compartmentalize Abstract Concepts into Classes?

- Your ability to compartmentalize classes from abstract concepts depends on experience and need of the situation. There is no hard-and-fast rule.
- Some rules of thumb:
  - Normally we have various entities and relationships (read “objects”) involved in a system.
  - Initially, tend to use single class. That is, if there is no differentiating properties among the entities, then no need to use inheritance at all.
  - If not possible, think about some common properties. Group them in the “root” class.
  - Think which entities differ mutually and which entities are subsets of which.
  - If one grouping does not work, try a different grouping.
  - Even if it works, still weigh other groupings.
  - Weigh the option of using inheritance vs **Composition** (to be discussed).

Composition

- E.g.: every car has an engine, every college has departments
  - So a car class will have a variable called engine, which may be another class.
- So the composition is: use of an instance variable that refers to other objects.
- It is reusing the code because otherwise we would need to copy the code of engine class to all car objects.

Inheritance vs Composition

- Inheritance: “is-a” relationship
- Composition: “has-a” relationship
- Both facilitate code reuse.
- Which one to use? Like many things in software engineering, this depends on the need, subject to experience of the programmer. “One size doesn’t fit all”.

How much inheritance is too much?

Software engineers say:

- Use inheritance when necessary.
- When using inheritance, stick to single inheritance if that works. Add multiple inheritance only if necessary.
- Unplanned inheritance makes the code bad.

Topics

Method overriding

- Syntax
- Motivation

Recap: Method Overriding

- When a method prototype in subclass is the same as a method prototype in superclass (which is inherited), then the method of subclass overrides the method of superclass.
- The said method of superclass is still inherited, but can’t be accessed unless we use the keyword “super”.
- Important: method overriding occurs only when the method prototypes are the same, otherwise method overloading will take place.

Method Overriding: Motivation

- Why use overriding?
  - When we use inheritance, oftentimes we need to implement different versions of a same “action” (which is implemented through method) in subclasses of the same level.
    - Real life example: a vehicle has an action of driving, whose implementation is different in road vehicles and water vehicles. Both are driven but the mechanism is different.
    - Coding example: a shape has an action of being drawn, whose implementation is different in circle and rectangle. Both are drawn but the mechanism is different.
  - So it is natural for these subclasses to put methods that have the same prototype but different course of actions.
    - E.g.: a method called `drive()` in both `road_vehicle` class and `water_vehicle` class, along with one in the `vehicle` class.
  - So is there any need to use method overriding here? Not yet.
  - Now, when we call the method `drive()` for different subclasses, we need to call the method using an object of respective type.

```
v.drive();      rv.drive();      wv.drive();
```

Method Overriding: Motivation (contd.)

- Why use overriding? (contd.)
  - A better way, however, would be to call using a generic name, e.g., `v.drive();` the reference variable `v` may refer to an object of type `vehicle` or `road_vehicle` or `water_vehicle`.
    - Why better? Because this provides a clearer and more natural and plausible interface in the sense that a superclass variable is referring to any subclass, and the type of the object (not the type of the variable) decides which version of the function will be called. Together this mechanism builds an easily comprehensible structure.
    - Also, the time for modifying the code is greatly minimized because we can write `v.drive()` in all places, and if we change only the object referred to by `v`, different versions of the method `drive()` will be executed. Take an example in the next slide.

Method Overriding: Motivation (contd.)

```
vehicle v = new vehicle();
road_vehicle rv = new road_vehicle();
water_vehicle wv = new water_vehicle();

if (random_event or user_input) {
  v.drive(); //1st call
  … //lots of lines of code
  v.drive(); //100th call
}
else if (random_event or user_input){
  rv.drive(); //1st call
  … //lots of lines of code
  rv.drive(); //100th call
}
else {
  wv.drive(); //1st call
  … //lots of lines of code
  wv.drive(); //100th call
}
```

Method Overriding: Motivation (contd.)

```
vehicle v = new vehicle();
road_vehicle rv = new road_vehicle();
water_vehicle wv = new water_vehicle();

if (random_event or user_input) {
  v.drive(); //1st call
  … //lots of lines of code
  v.drive(); //100th call
}
else if (random_event or user_input){
  rv.drive(); //1st call
  … //lots of lines of code
  rv.drive(); //100th call
}
else {
  wv.drive(); //1st call
  … //lots of lines of code
  wv.drive(); //100th call
}
```

```
vehicle v = new vehicle();
road_vehicle rv = new road_vehicle();
water_vehicle wv = new water_vehicle();

if(random_event or user_input){
 v = v;
}
if (random_event or user_input){
  v = rv;
}
else {
  v = wv;
}

v.drive(); //1st call
… //lots of lines of code
v.drive(); //100th call
```

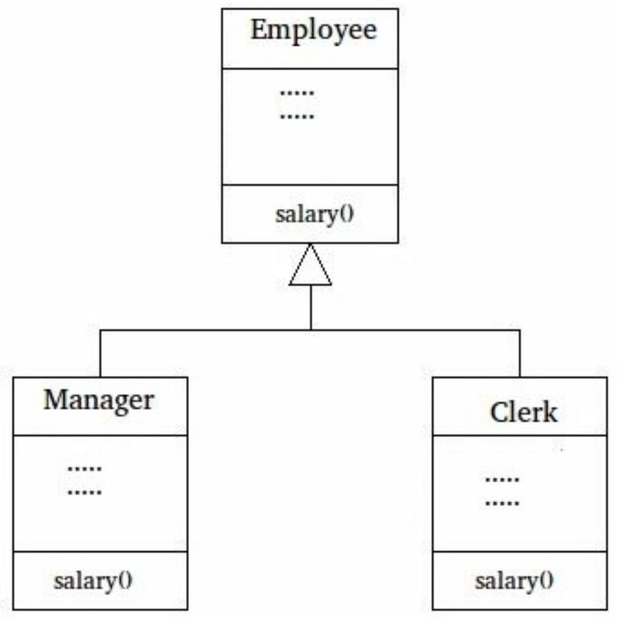
Method Overriding: Motivation (contd.)

- Why overriding: another perspective
  - From the previous slide we know that it is a good idea to use a single variable to refer to multiple objects so that a common member can be accessed in a generic way (`v.drive()` in our example).
  - Now, we can’t do that unless we establish some “relationship” among the two objects in question.
  - So we add a superclass for both the classes with the intention to use the superclass variable to access both the subclasses.
  - But that is not sufficient because a superclass variable doesn’t have any knowledge of the members of subclasses (recall this topic from a few slides back).
  - So we add in the superclass the method that is common to the subclasses (which we want to execute, in our example, the `drive()` method).
- Method overriding is thus yet another means to achieve polymorphism.
  - It involves both inheritance and polymorphism, whereas method overloading may or may not involve inheritance.

Method Overriding: Another Example

```
class Employee {
    public static int base = 10000;
    int salary()  {          return base;     }
}
class Manager extends Employee {
    int salary()     {         return base + 20000;      }
}
class Clerk extends Employee {
     int salary()     {         return base + 10000;      }
}
class Main {
    static void printSalary(Employee e)
    {         System.out.println(e.salary());     }

    public static void main(String[] args)
    {
        Employee mangr = new Manager();
        System.out.print("Manager's salary : ");       printSalary(mangr);
        Employee crlk = new Clerk();
        System.out.print("Clerk's salary : ");         printSalary(crlk);
    }
}
```



https://www.geeksforgeeks.org/overriding-in-java/(modified)

Method Overriding: Early Binding Vs. Late Binding

- Compile-time polymorphism (aka static polymorphism/binding, early binding).
  - Here the version of a polymorphic function to be invoked is selected during compile time.
  - E.g.: the particular version (out of multiple versions) of an overloaded function to be called is resolved by the compiler.
    - Caveat: unless the choice depends on user input.
- Run-time polymorphism (aka dynamic polymorphism/binding, late binding).
  - Here the version of a polymorphic function to be invoked is selected during execution time.
  - E.g.: the particular version (out of multiple versions) of an overridden function to be called is resolved by the JVM. This mechanism is called Dynamic Method Dispatch.
- Static binding uses type (of class) information for binding at compile time while dynamic binding uses instance of class (i.e., physical object's properties) to resolve calling of method at run-time.

Method Overriding: Early Binding Vs. Late Binding

- OK, but why resort to late binding (i.e., runtime polymorphism)? Any benefit?
  - Resolving which version of an overridden method to execute during compile time is not easy for a compiler. So this task is left to JVM to be done at runtime.
    - Details of this explanation requires an understanding the technical details of compiler and interpreter (JVM) which is out-of-scope of this course.
    - For some details: http://www.artima.com/insidejvm/ed2/index.html
  - Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
    - The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

End of Lecture 10.