Object-oriented programming

Lecture #7: Inheritance

Additions to Last Lecture

- Constructors: if no constructor is defined, the compiler will create a default one for us (same for destructor)
- Access Modifiers:
  - The access modifiers work on **class level**, and not on **object level**
  - That is, two objects of the same class can access each others private members

Example (Date)

```
class Date {
   private:
     int month, day, year;
   public:
     Date(int mo, int dy, int yr) {
                  month = mo; day = dy; year = yr; }
     Date( Date & d) {
        month = d.month; day = d.day; year = d.year; //WHY
     }
};
```

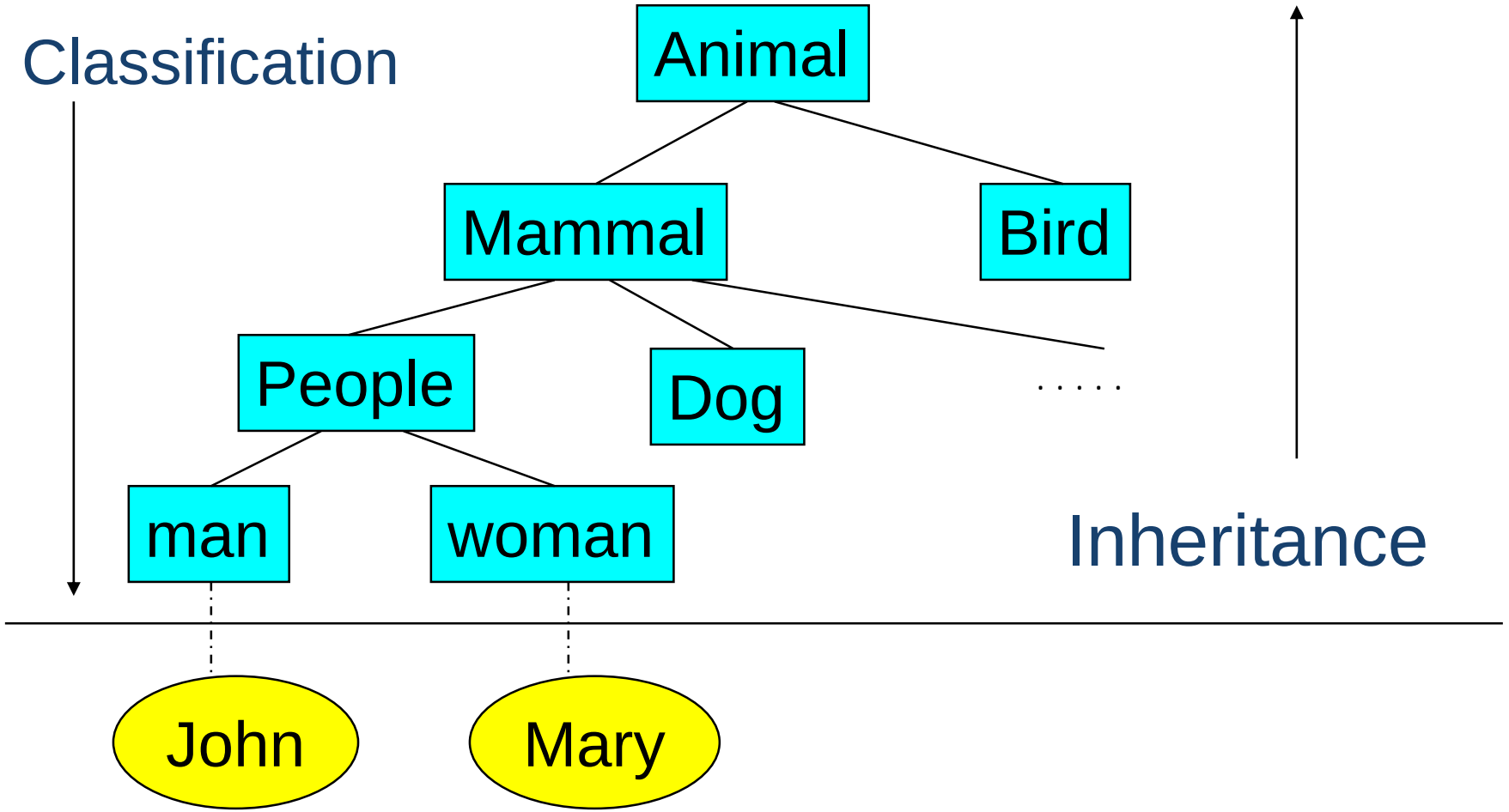
WHY???

- The private modifier enforces Encapsulation principle.
- The idea is that 'outer world' should not make changes to Person internal processes because Person implementation may change over time (and you would have to change the whole outer world to fix the differences in implementation - which is nearly to impossible).
- When instance of Person accesses internals of other Person instance - you can be sure that both instances always know the details of implementation of Person. If the logic of internal to Person processes is changed - all you have to do is change the code of Person.

Outline

- Classification
- Sharing
- Inheritance
- Subclass
- The Categories of Inheritance
- Abstract Class

Classification/Inheritance



Classification

- Classification arises from the universal need to describe uniformities of collections of instances
- Classification is a basic idea for understanding the concept inheritance

Classification/Inheritance

- Commonality
  - The base class captures the common information (attributes) and features (operations) of the derived classes
- Customization
  - An existing class is used to create a customized version of the class
- Common Design Interface
  - A base class may define the design requirements for its derived classes by specifying a set of *member* functions that are required to be provided by each of its derived classes

Sharing

- Sharing is ubiquitous in real-world, thus it is important in object-orientation
- Inheritance is a technique that promotes sharing
- Inheritance means that new classes can be derived from existing classes
- A subclass (derived class) inherits the attributes and the operations of a superclass (base class); but a subclass can define additional operations and attributes

Sharing

- Can be seen a a specialization mechanism
  - Instances of a subclass are specializations (additional state and behavior) of the instances of a superclass
- Can also be seen as generalization mechanism
  - Instances of a superclass generalizes the instances of a subclass

Sharing

- Is also the basic idea of inheritance
- The ability of one class to share the behavior of another class without explicit redefinition
- An approach which allows classes to be created based on an old class

Three Dimensions of Sharing

- Static or dynamic sharing:
  - At times, sharing patterns have to be fixed. This can be done at object creation (static) or when an object receives a message (dynamic)
- Implicit or explicit sharing:
  - The programmer directs the patterns of sharing (explicit) or the system does it automatically (implicit)
- Per object or per group sharing:
  - Behaviors can be specified for an entire group of objects or can be attached to a single object

Purposes of Inheritance

1. Reusing existing design reduces software development and testing costs
2. Modular Design: A new class inherits the data and functionalities of an existing class

What is Inheritance?

- **Inheritance** is a mechanism for expressing similarity
- **Inheritance** is a natural property of classification
- **Inheritance** is a mechanism that allows (a class) A to inherits properties of (a class) B

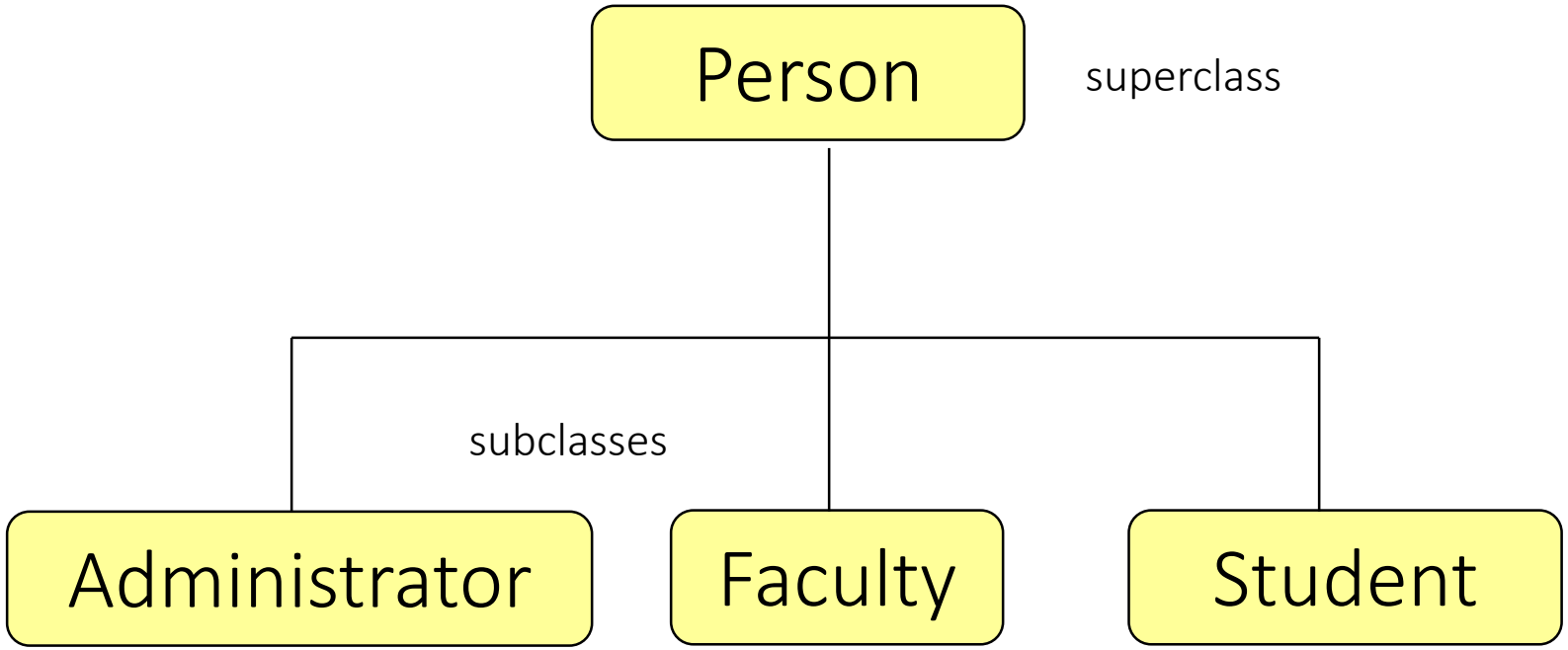
Inheritance in OOP

- Assume “A inherits from B”. Then objects of class A have access to attributes and methods of class B without the need to redefine them

Superclass/Subclass

- If class A inherits from class B, then B is called **superclass** of A. A is called **subclass** of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share similar (signature) behavior as objects of the superclass

Example


Person
superclass
subclasses
Administrator
Faculty
Student

Subclasses

- Subclasses refer to not only inheritance of specification but also inheritance of implementation and so can be viewed as reusability mechanism

Important Aspects of Subclasses

- Modifiability
  - The degree of modifiability determines how **attributes** and **methods** inherited from a superclass can be modified in a subclass. In this context, an approach of distinguishing modifiability of the object states (attributes) and the object behaviors (operations) are used

Attributes

1. No redefinition: modification is not allowed
2. Arbitrary redefinition: redefinition without constrained is allowed
3. Constrained redefinition: the domain of attributes is constrained
4. Hidden redefinition: definitions of attributes are hidden in subclass to avoid conflicts

Operations

1. Arbitrary redefinition: all changes to operations are allowed
2. Constrained redefinition: Parts of the signature of methods in the subclass have to be subtypes of the parts of the methods of the superclass

- This is important in overriding and overloading of methods

Naming Conflicts

- Conflicts between a superclass and a subclass
  - When attributes or methods defined in a subclass has the same names as attributes or methods defined in the superclass
  - This is resolved by **overriding**

Categories of Inheritance

- Whole/Partial Inheritance
  - Whole Inheritance is when a class inherits all the properties and operations from its superclass
  - Partial Inheritance is when only some properties are inherited while others are suppressed

Categories of Inheritance

- Default Inheritance: inherited properties and constraints can be modified
- Strict inheritance: doesn’t allow the user to modify inherited properties or constraints

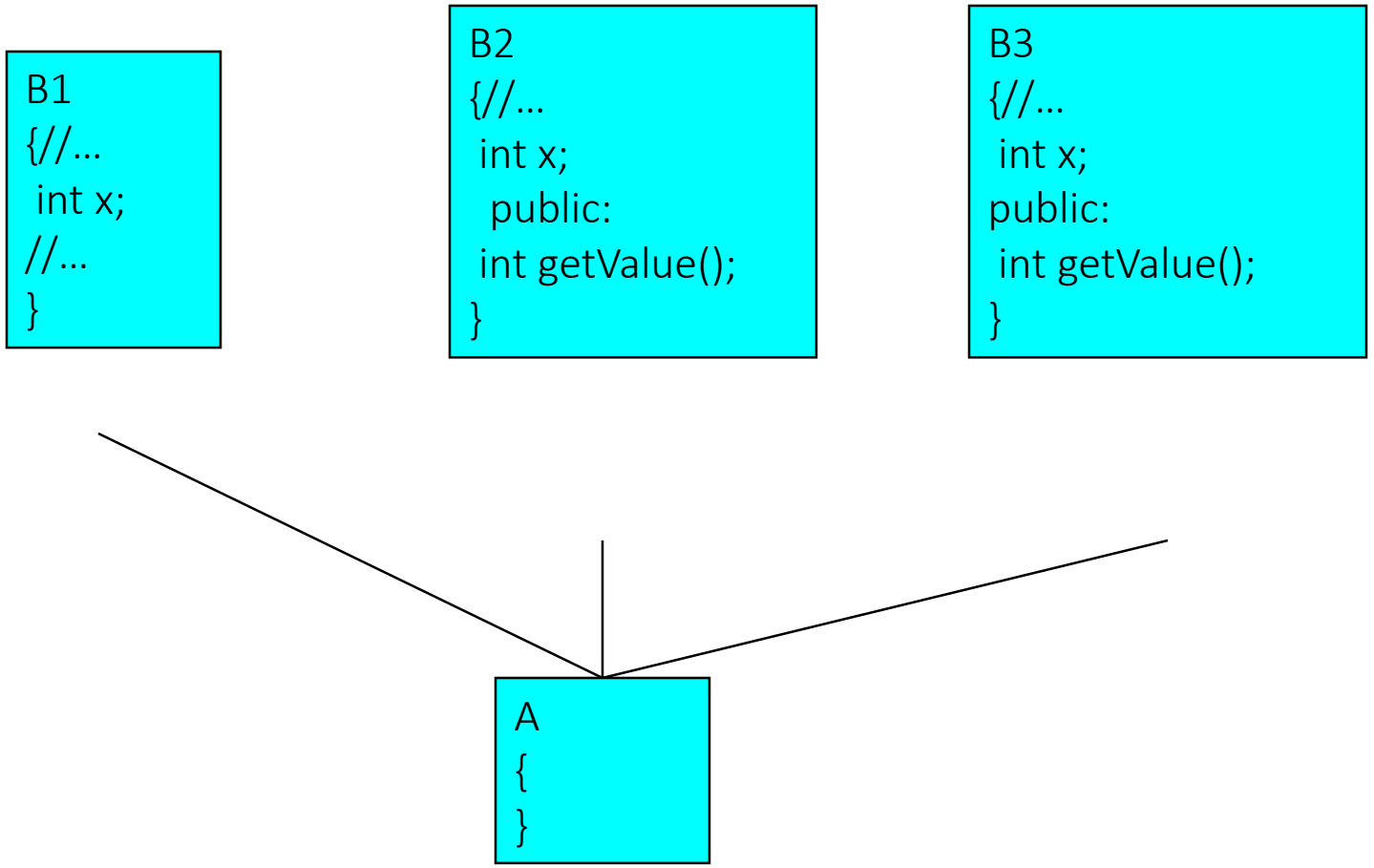
Categories of Inheritance

- Single (simple) inheritance:
  - A class can inherit from one superclass only. This means the inheritance hierarchy forms a tree
- Multiple inheritance
  - A class can have more than one superclass

Multiple Inheritance

- **Naming Conflicts:** if attributes or methods defined in one superclass have the same name as attributes or methods defined in another superclass

Naming Conflicts



Conflict Resolution

1. Using the order of the superclass

- If there are more attributes or methods with the same name in different superclasses, the ones occurring in the first class in the list of superclasses are inherited (done by the compiler)
- If the order is from left to right, then x means x of B1, and getValue() means the getValue() of B2

Conflict Resolution

2. Determined by the user

- The user can inherit conflicting attributes or methods, but has to explicitly rename conflicting attributes or methods in the subclass (done by the user)
- B1::x; or B2::x; or B3::x;
- B2::getValue(); or B3::getValue();

Abstract Class

- A class without any instance
- Mammal is an abstract class: only for classification and inheritance. Never make objects of the class.

Abstract Class

- It is also possible to have a classification which does not have any (executable) code behind it

Abstract Class

- Class A is called an **abstract class** if it is used only as a superclass for other classes. Class A only specifies properties. It is not used to create objects. Derived classes must define the properties of A.

Reuse

- Many take inheritance as a reusing tool
- In this case, we can take inheritance as a code reusing tool

Inheritance/Classification

- Classification implies inheritance (whole and simple inheritance)
- Inheritance may effect clarity of classification (for multiple inheritance and for part inheritance)

Declaring Derived Classes

```
class derived_class_name :
access_specifier base_class_name
{ /*...*/ };
```