Object-Oriented Programming

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2017

Overview

Primary OOP features

- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects.
- **Inheritance**: allowing code to be reused between related types.
- **Polymorphism**: allowing an object to be one of several types, and determining at runtime how to "process" it, based on its type.

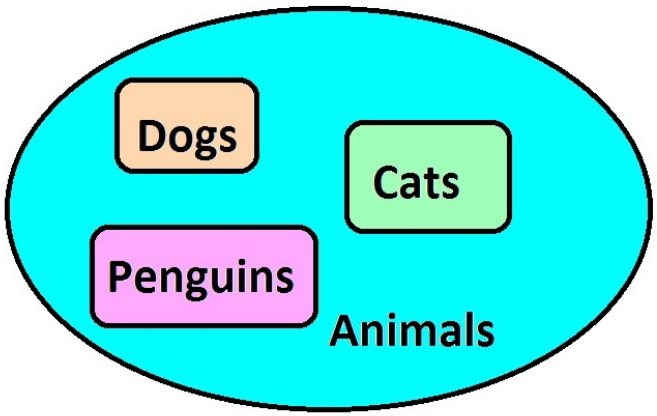
Inheritance I

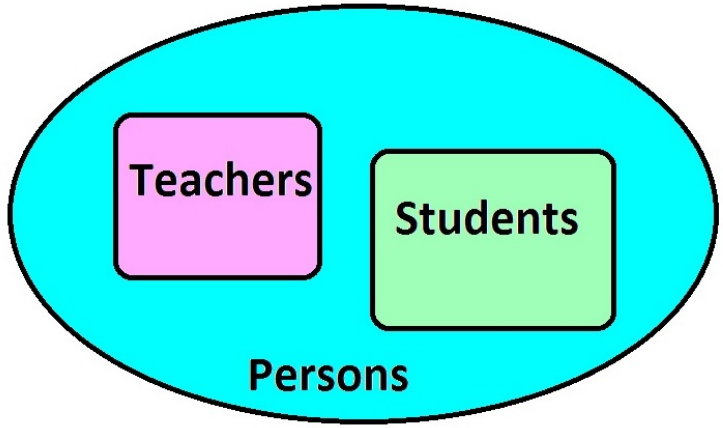
- Allows defining a new class (subclass) by using the definition of another class (superclass).
- Inheritance makes code reusability possible.
- Reusability refers to using already existing code (classes).
- The time and effort needed to develop a program are reduced, the software is more robust.

Inheritance II

- Through inheritance, new classes cab be derived from already existing ones.
- The existing class is not modified.
- The new class can use all the features of the old one and add new features of its own.
- Inheritance can be used if there is a **kind of** or **is a** relationship between the objects.

Example





Simple inheritance - Derived classes I

- Inheritance requires at least two classes: a *base class* and a *derived class*.

- If B and D are two classes,
  - D *inherits from* B or
  - D *is derived from* B or
  - D *is a specialization of* B
- means that:
  - class D has all variables and methods of class B;
  - class D may redefine methods of class B;
  - class D may add new members besides the ones inherited from B.

Simple inheritance - Derived classes II

- If class D inherits from class B then:
  - an object of class D includes all member variables of class B;
  - the member functions of class B can be applied to objects of class D (unless they are hidden).

Syntax

```
class D: public B
{
// ...
};
```

Simple inheritance - Derived classes III

Example

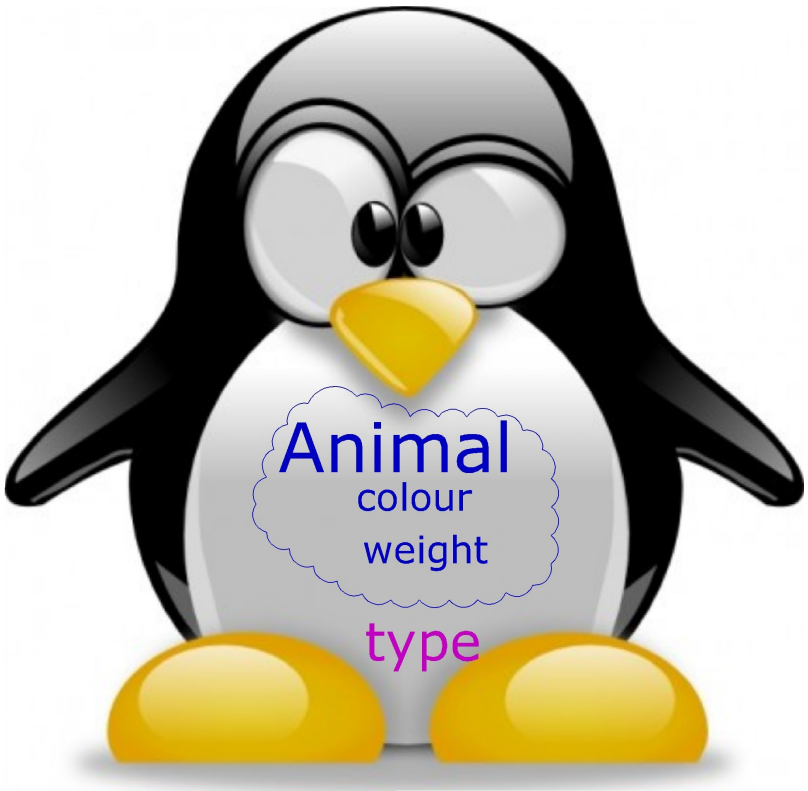
```
class Animal
{
protected:
    std::string colour;
    double weight;
//...
};

class Penguin: public Animal
{
private:
    std::string type;
//...
};
```

Simple inheritance - Derived classes IV

DEMO

Class derivation (Animal - Penguin, Dog) (*Lecture5_demo*).



Simple inheritance - Derived classes V

Terminology

- class B = superclass, base class, parent class.
- class D = subclass, derived class, descendent class.
- *inherited member (function, variable)* = a member defined in B, and used unchanged in D.
- *redefined member (overridden)* = defined in B and D.
- *added member (new)* = defined only in D.

Access modifiers I

Access modifiers define where the members of a class (fields or methods) can be accessed from.

- public: public members can be accessed from anywhere.
- private: private members can be accessed from within the class or from friend functions or classes.
- protected: protected members can be accessed from within the derived classes; protected acts just like private, except that inheriting classes have access to protected members, but **not** to private members. Friend functions or classes can access protected members.

Access modifiers II

Access	public	protected	private
Class	Yes	Yes	Yes
Derived class	Yes	Yes	No
Client code	Yes	No	No

Access control I

- Public inheritance:
  - The access rights of the members of the base class are not changed.

```
class A: public B { ... }
```

- Protected inheritance:
  - Inherited public or protected members from the base class become protected members in the derived class.

```
class A: protected B { ... }
```

- Private inheritance:
  - Inherited public or protected members from the base class become private members in the derived class.

```
class A: private B { ... }
```

Access control II

Inheritance type	public	protected	private
Base access specifier	**Derived access specifier**		
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private

Special member functions and inheritance

- Some functions will need to do different things in the base class and the derived class.
- These special functions cannot be inherited.
- **Constructors**: derived class constructor must create different data from base class constructors.
- **Assignment operator**: in the derived class, this operator must assign values to the derived class data.
- **Destructors**

Constructors and destructors for derived classes I

- Constructors and destructors are not automatically inherited.
- Constructors in the derived class need to invoke a constructor from the base class.
- If no constructor is not explicitly invoked, the *default constructor* from the base class is invoked automatically.
- If there are no default constructors $\rightarrow$ compiler error.
- **?** How is it possible to *not* have a default constructor?

Constructors and destructors for derived classes II

- When an object of a derived class is created, the constructor of the base class is called first and then the constructor of the derived class.
- The destructor of the base class is automatically invoked by the destructor of the derived class.
- When an object of a derived class is destroyed, the destructor of the derived class is called first and then the destructor of the base class.

Constructors and destructors for derived classes III

Object creation in derived classes

- Creation:
  1. allocate memory for member variables from base class;
  2. allocate memory for member variables from derived class;
  3. a constructor is selected and called to initialize the variables from the base class;
  4. a constructor is selected and called to initialize the variables from the derived class.
- Destruction:
  1. destructor call for derived class;
  2. destructor call for base class.

Constructors and destructors for derived classes IV

DEMO

Creation and destruction in derived classes (*Lecture5_demo*).

Liskov substitution principle

- If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T.
  (Barbara H. Liskov and Jeannette M. Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, 1994.)

- An object of the derived class can be used in any context expecting an object of the base class (upcast is implicit).

DEMO

Substitution principle (*Lecture5_demo*).

Pointers and inheritance

- If class D publicly inherits from class B, then a pointer to D can be assigned to a variable of type pointer to B.
- A pointer to an object of type B can carry the address of an object of type D.
- E.g.: A pointer to an animal can point to objects of type Animal, Dog and Penguin (all dogs and penguins are animals).

DEMO

Pointers and inheritance (*Lecture5_demo*).

Method overriding I

- A derived class may override (redefine) some methods of the base class.
- In defining derived classes, we only need to specify what is different about them from their base classes (programming by difference).
- Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class.

Method overriding II

- Use the scope resolution operator **::** to access the overridden function of base class from derived class.
- Overriding $\neq$ overloading. **?** What is the difference?

DEMO

Overriding the *toString* method. (*Lecture5_demo*).

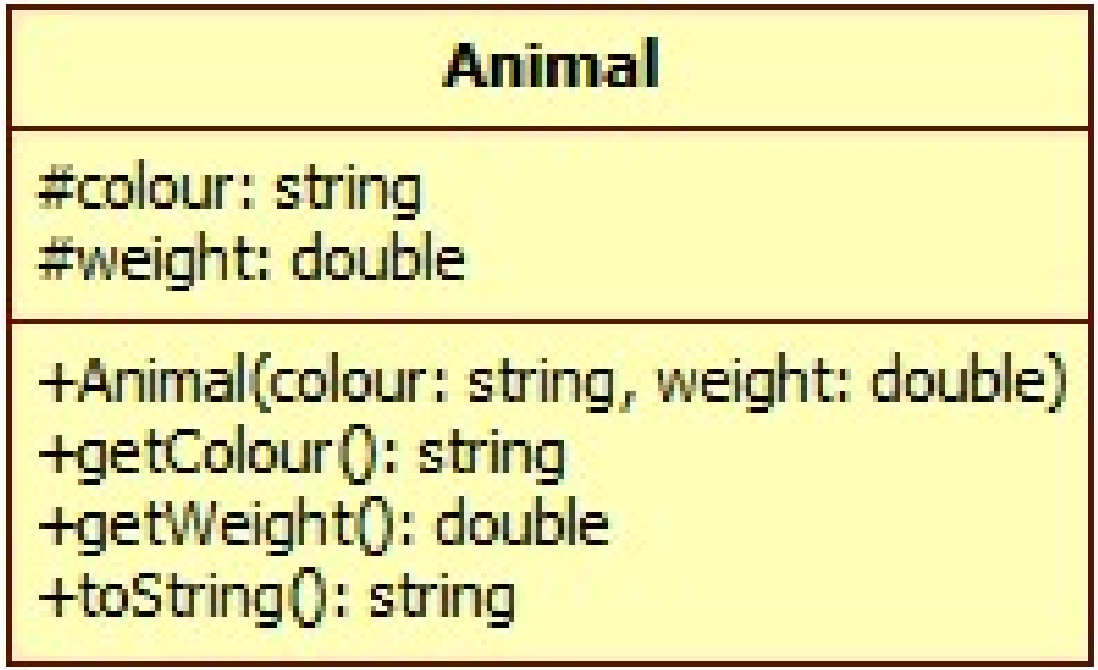
UML

- UML - Unified Modeling Language.
- UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- UML is the standard notation for software architecture.
- It is language independent.

UML class diagrams I

- A UML class diagram specifies the entities in a program and the relationships among them.
- It contains and specifies:
  - class name
  - variables (name, type)
  - methods (name, parameter types, return type)
- private members are denoted by -;
- public members are denoted by +;
- protected members are denoted by #.

UML class diagrams II



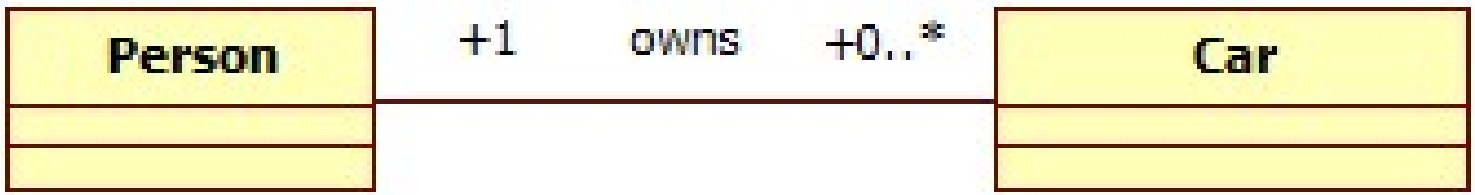
Associations I

- UML associations describe relationships of structural dependency between classes.
- An association may have:
  - a role name;
  - a multiplicity;
  - navigability (uni/bi-directional).

Associations II

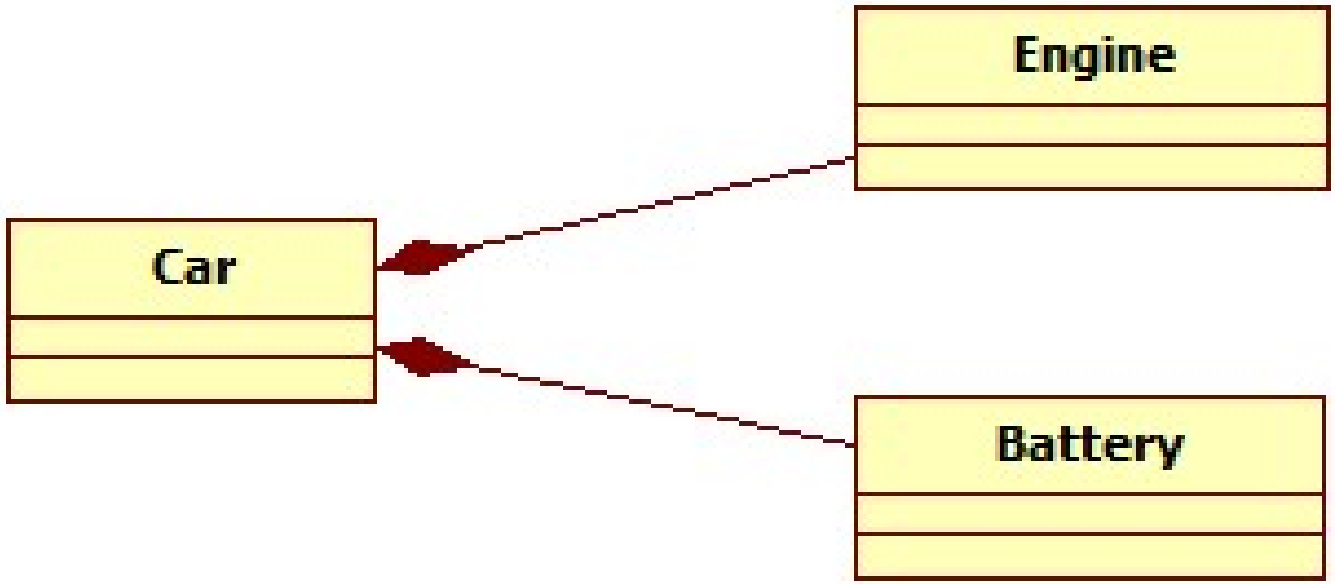
Association types

- **Association** (*knows a*) - is a reference based relationship between two classes. A class A holds a class level reference to another class B.



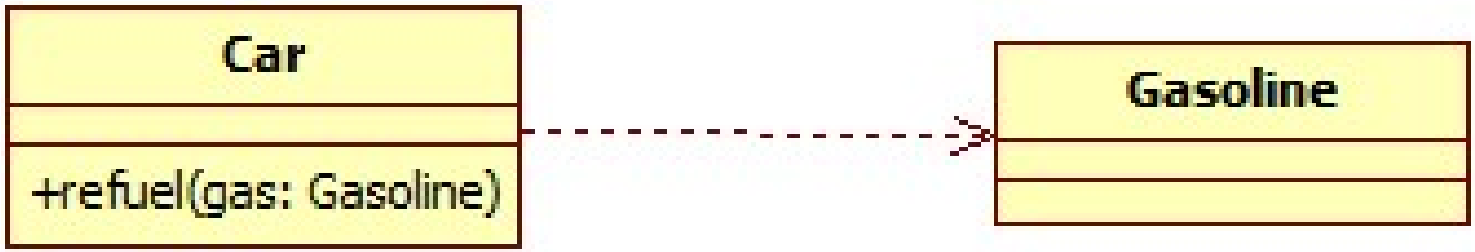
Associations III

- **Composition** (*has a*) - when class B is composed by class A, class A instance owns the creation or controls lifetime of instance of class B. When class A instance is destructed, so is the class B instance.



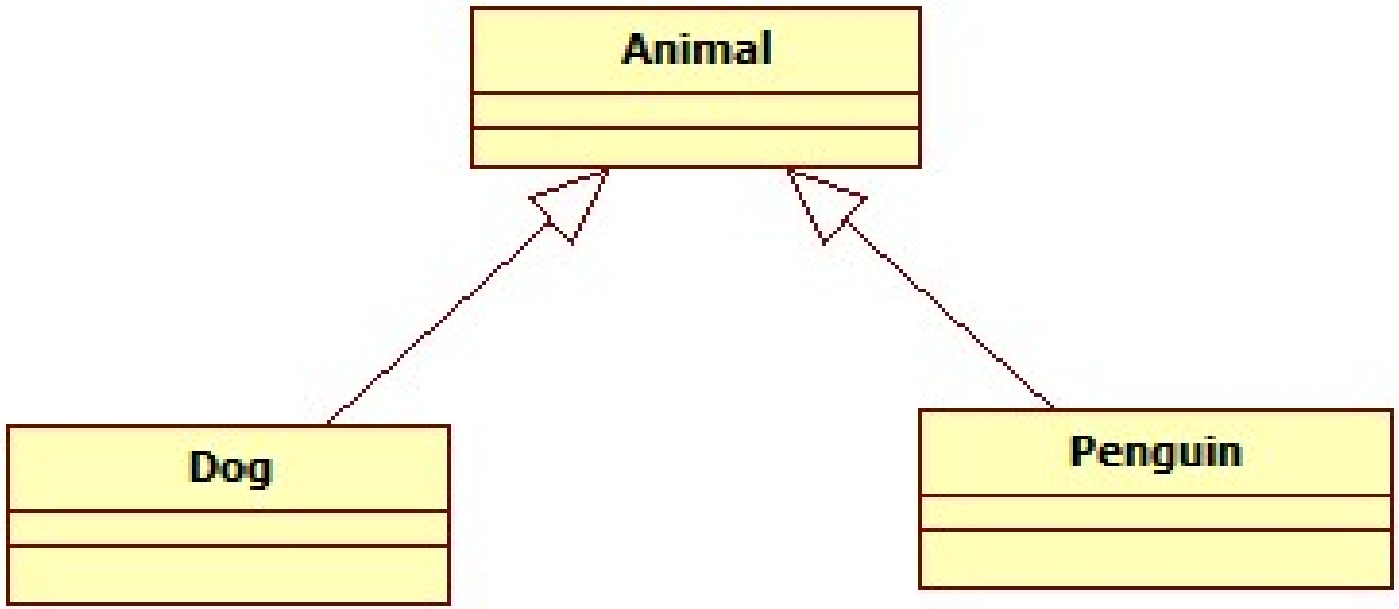
Associations IV

- **Dependency** (*uses a*) - when class A uses a reference to class B, as part of a particular method (parameter or local variable). A modification to the API of class B reference may influence class A.



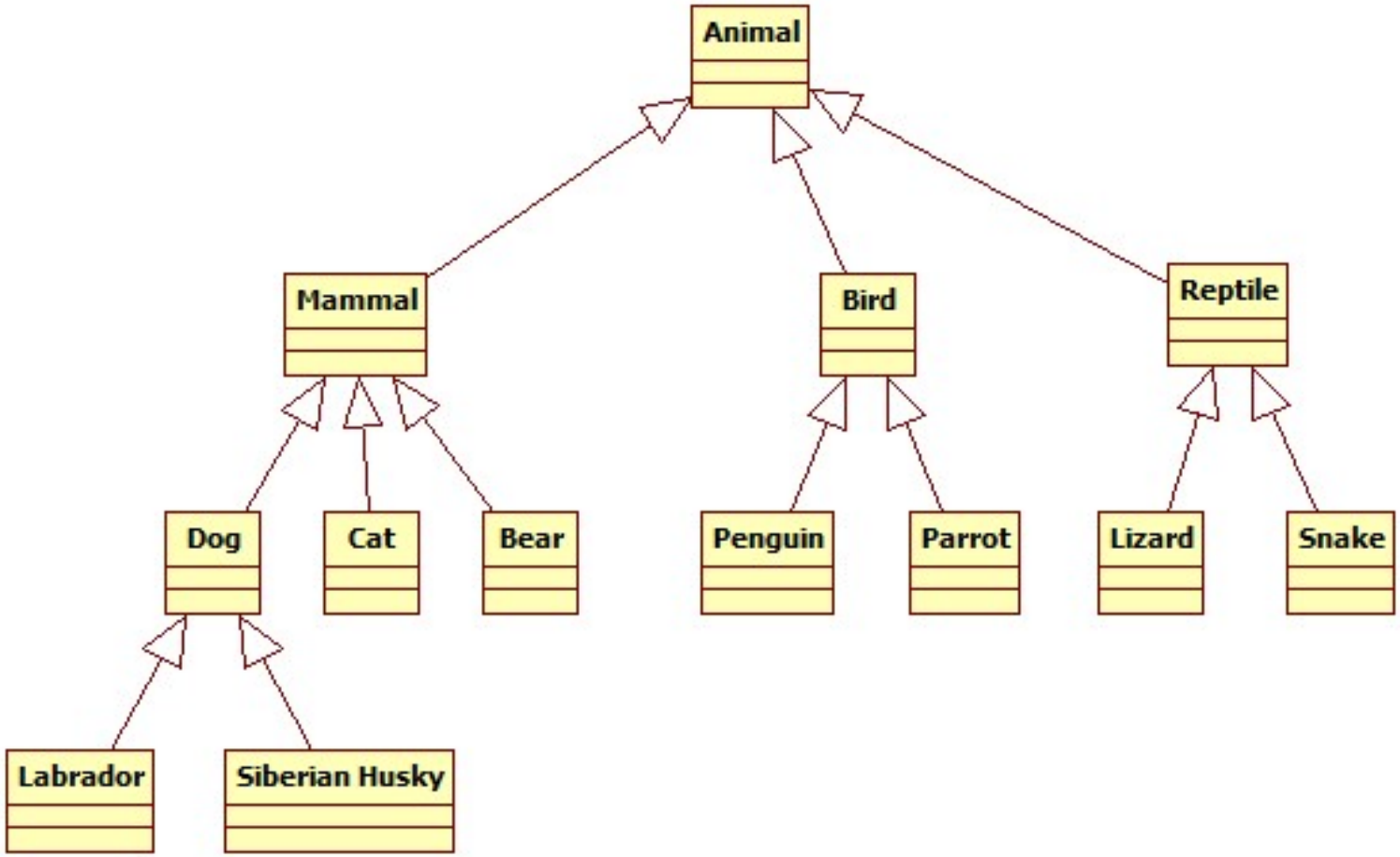
Associations V

- **Inheritance** (*is a*) - every instance of the derived class **is an** instance of the base class.



Associations VI

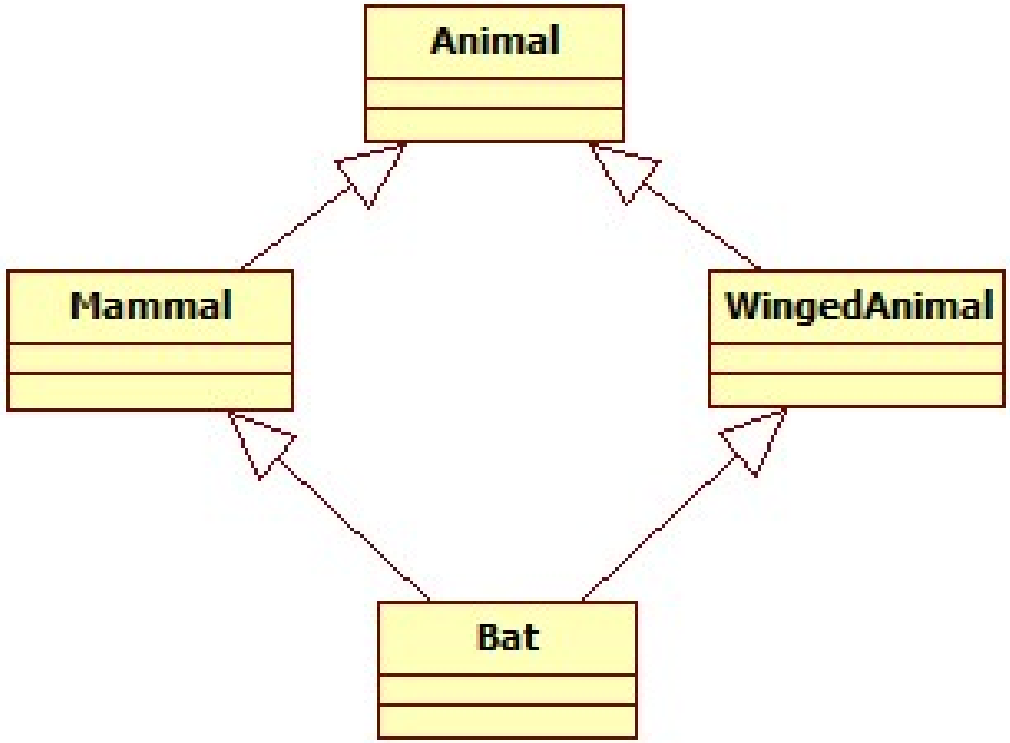
- Inheritance allows us to define hierarchies of related classes.



Multiple inheritance I

- Unlike many object-oriented languages, C++ allows a class to have multiple base classes.
- The class will inherit all the members from all the base classes.
- Multiple inheritance can be dangeuros:
  - the same field/method could be inherited from different classes;
  - the situation of repeated base classes might arise.
- **In general, you should avoid multiple inheritance.**

Multiple inheritance II



Multiple inheritance III

Problems with multiple inheritance

- **Ambiguity**: multiple base classes contain a function with the same name.
- **Diamond problem**: if a method from class Animal was overriden in both classes (Mammal and WingedAnimal), which of the two versions should be inherited?