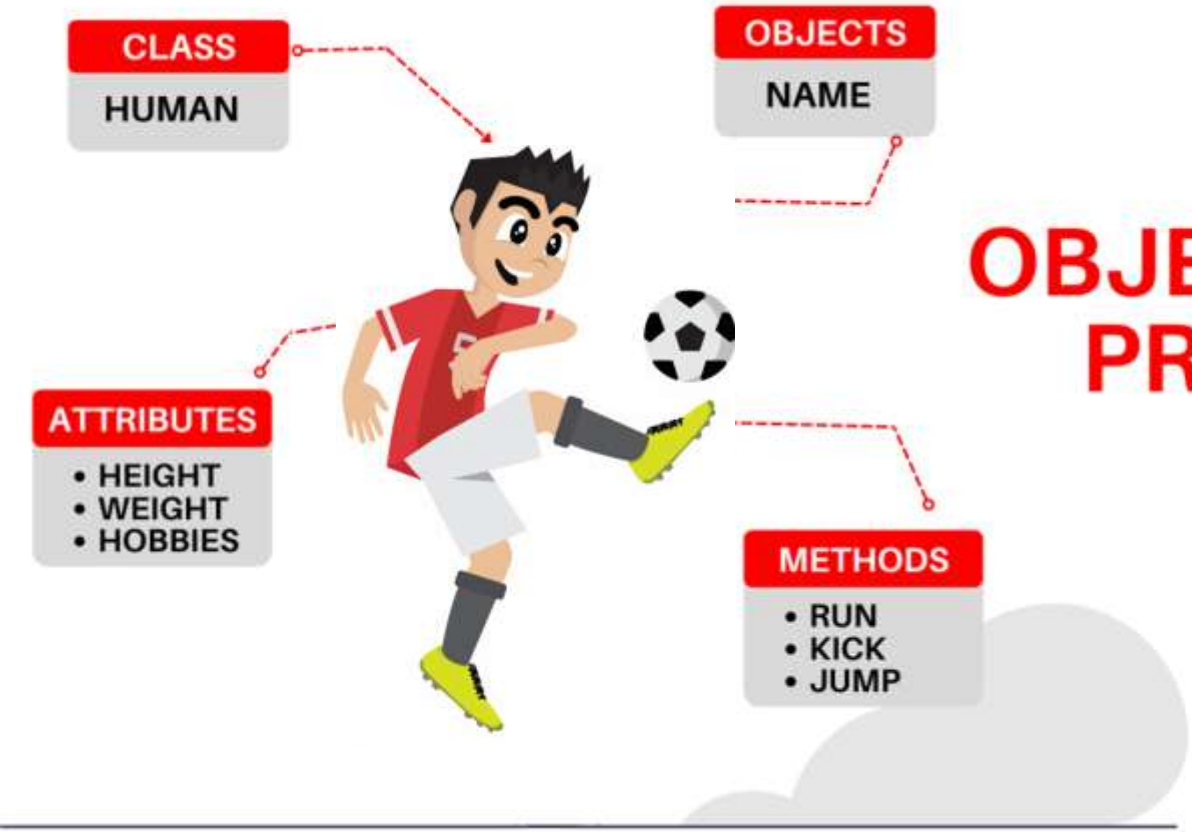
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

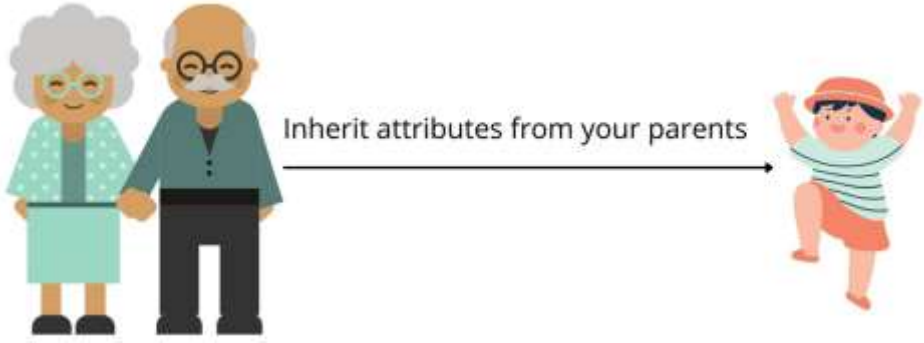
Summer 2023

Pir Sami Ullah Shah

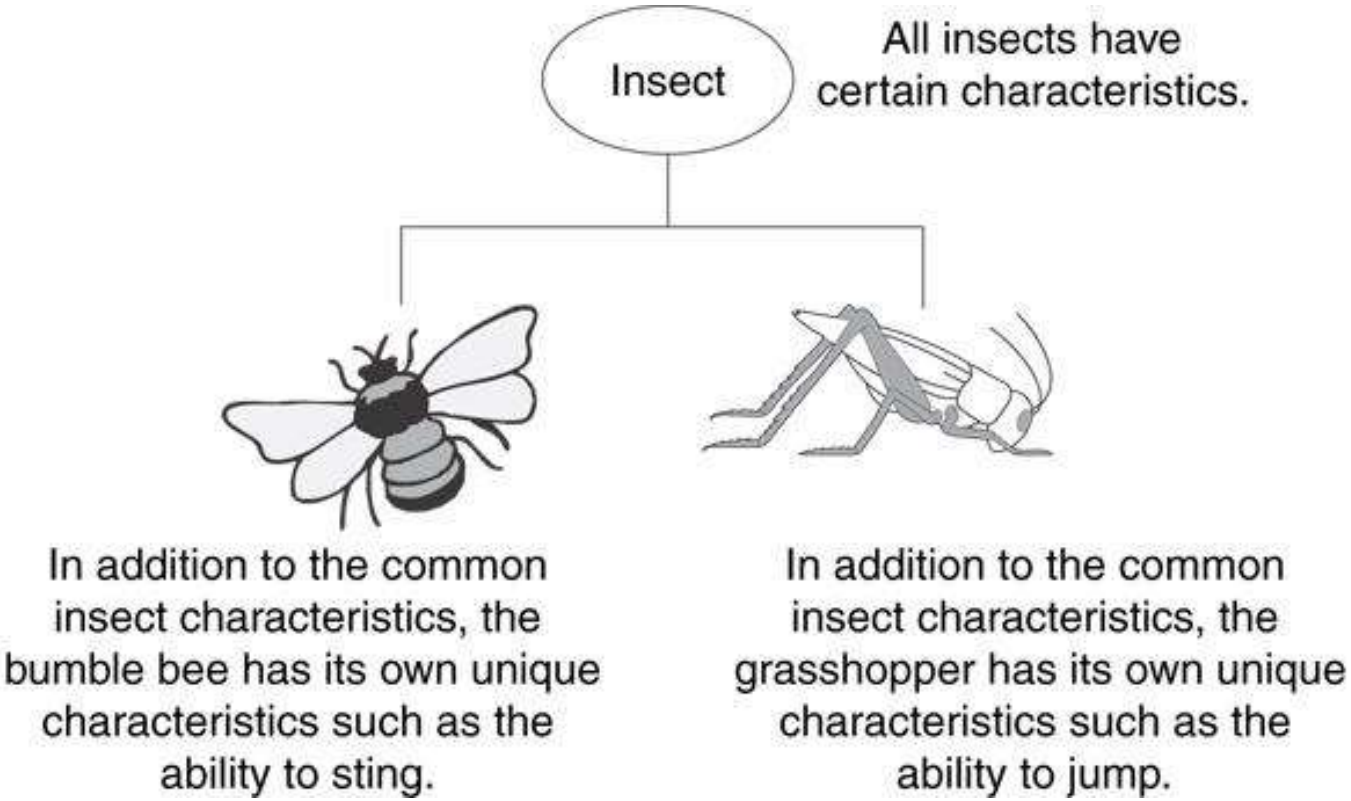
Lecture # 9 Inheritance

What Is Inheritance?

- One of the most powerful features of OOP
- Provides a way to create a new class from an existing class
- The new class inherits all the capabilities of the existing class and can also add capabilities of its own.
- The base class is unchanged by this process.
- The new class is a ***specialized version*** of the existing class



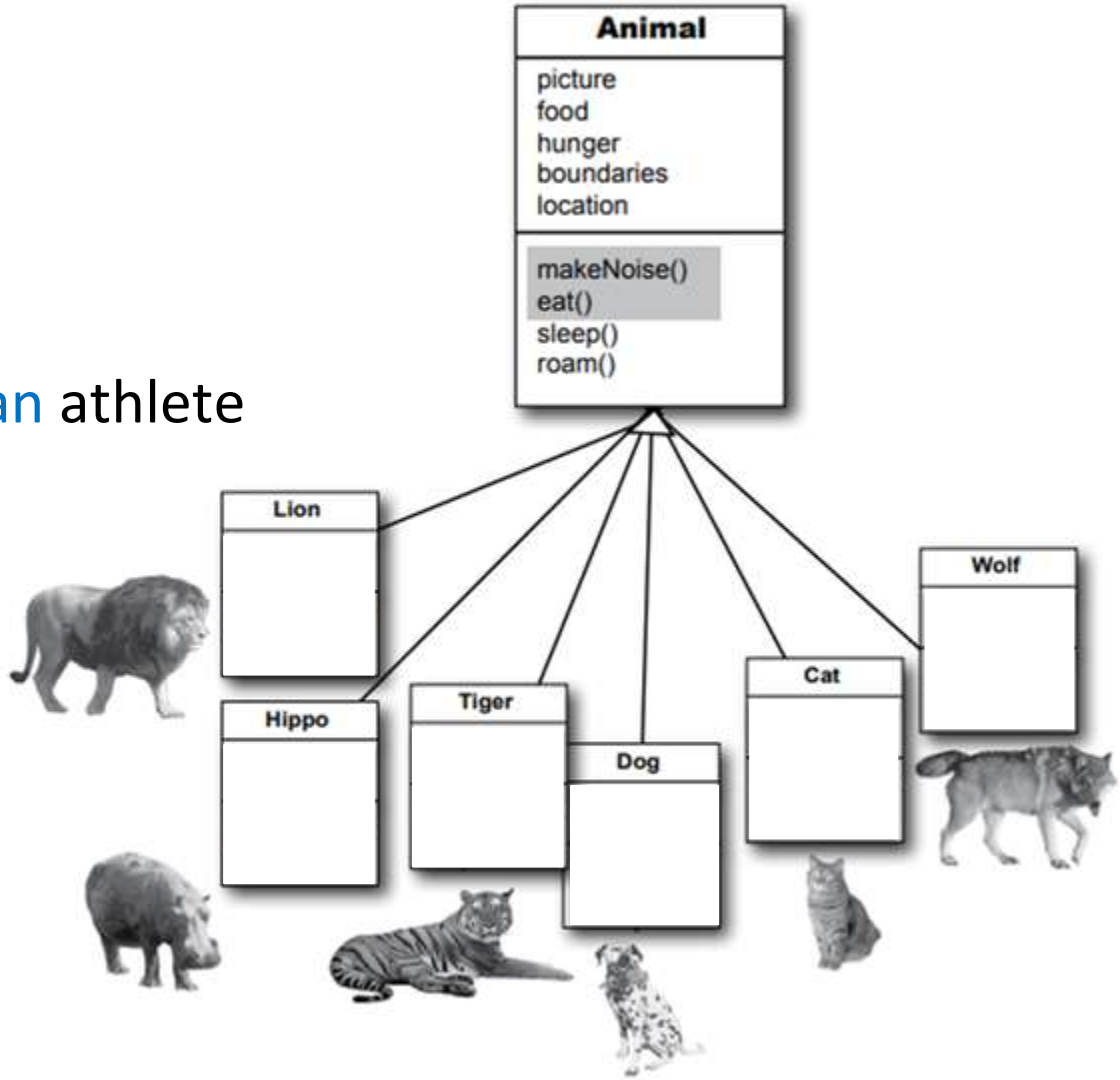
Example: Insects



- Insect is ***generic***
- Bee and grasshopper are ***specific***

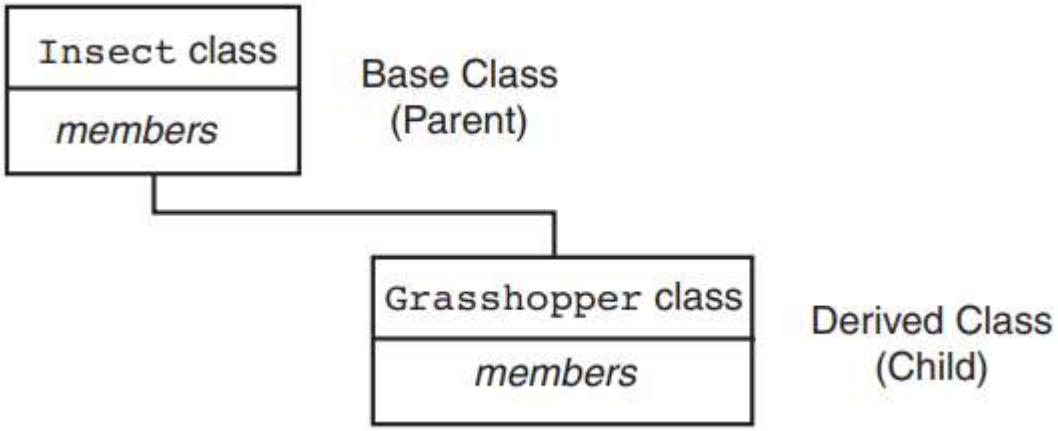
The “IS - A" Relationship

- Inheritance establishes an “***IS - A***" relationship between classes.
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete



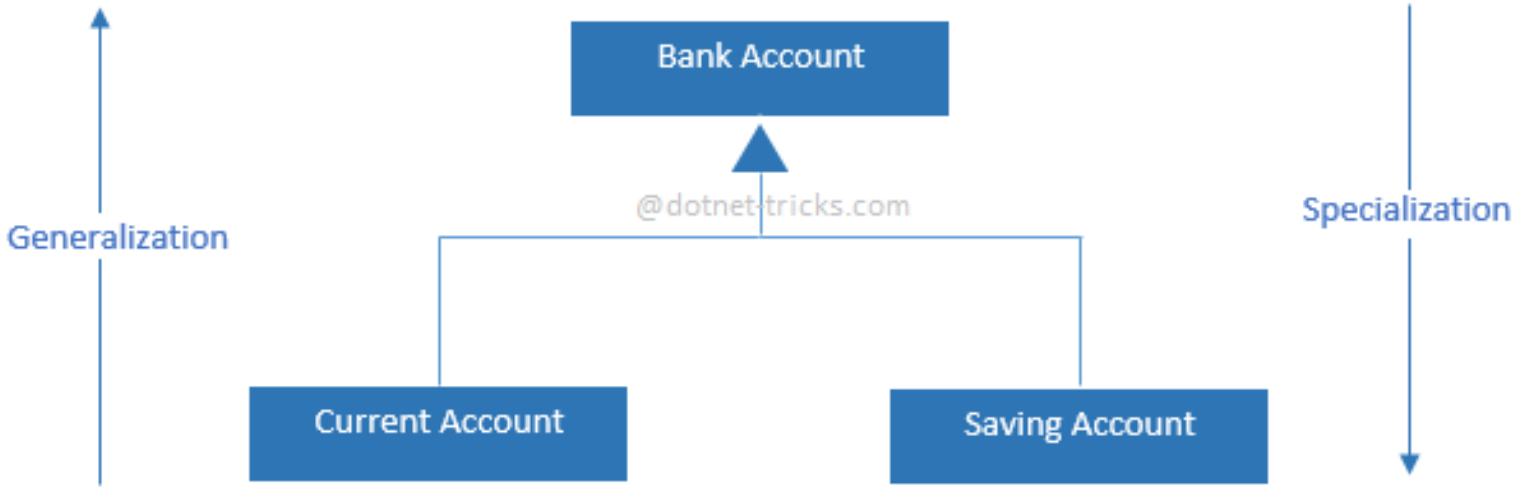
Introduction - Inheritance

- Existing classes are called base classes
- New classes are called derived classes

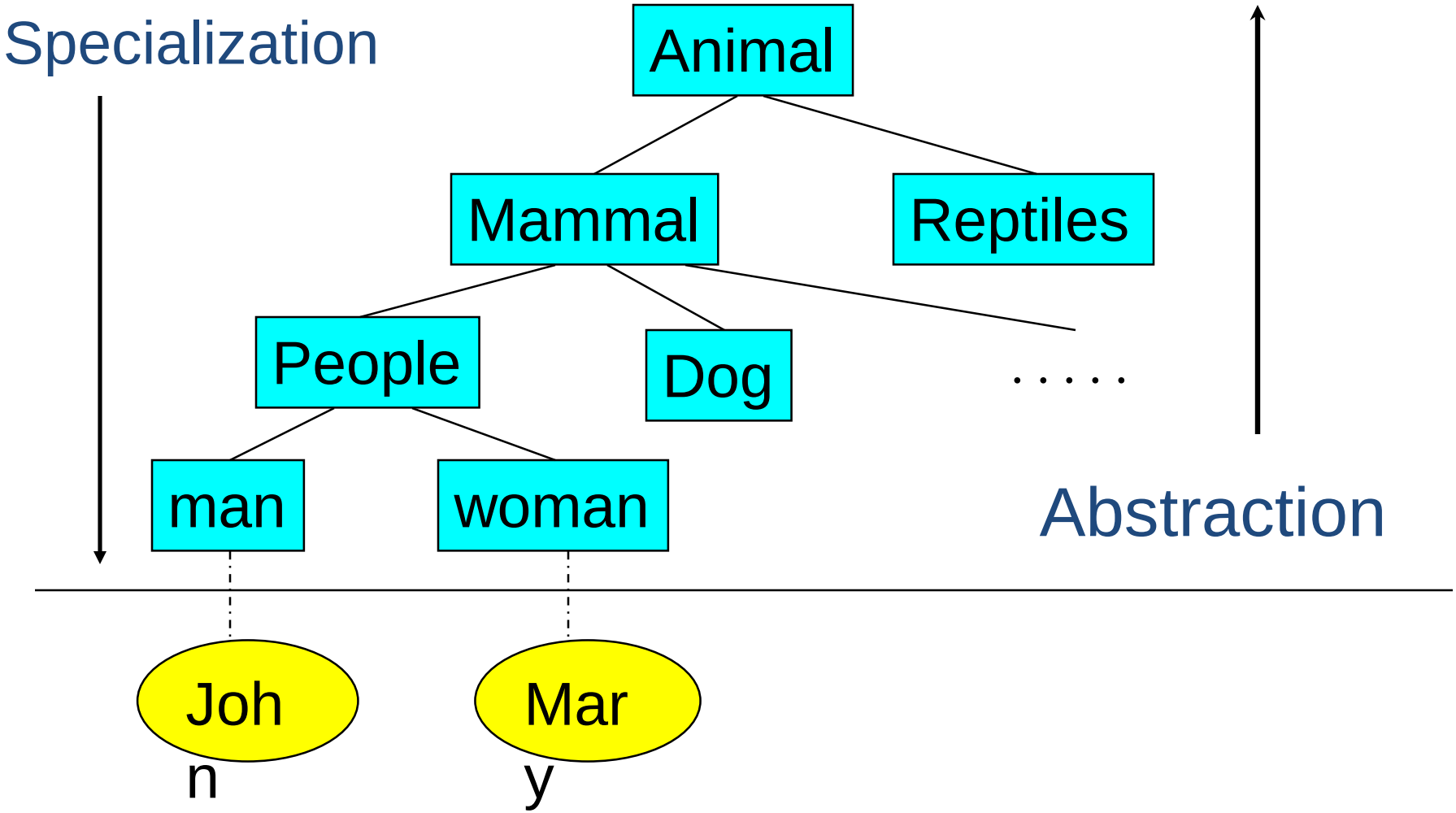


Introduction - Inheritance

- Objects of derived classes are more specialized as compared to objects of their base classes



Animals: Class's hierarchy



Inheritance Examples

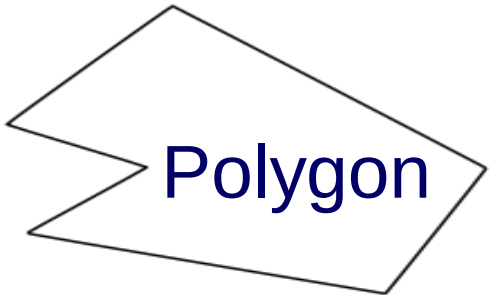
Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Why Inheritance?



Rectangle


```
class Rectangle{
 private:
         int numVertices;
         float *xCoord, *yCoord;
 public:
void set(float *x, float *y, int nV);
float area();
   };
```

```
class Polygon{
  private:
      int numVertices;
      float *xCoord, *yCoord;
  public:
void set(float *x, float *y, int nV);
};
```

```
class Triangle{
  private:
      int numVertices;
      float *xCoord, *yCoord;
  public:
void set(float *x, float *y, int nV);
float area();
};
```

Why Inheritance?

- Inheritance provides us a mechanism of software reusability which is one of the most important principles of software engineering
- Show similarities
- Easy modification by performing modification in one place
- Avoid redundancy, leading to smaller and more efficient model, easier to understand

Inheritance – Terminology and Notation

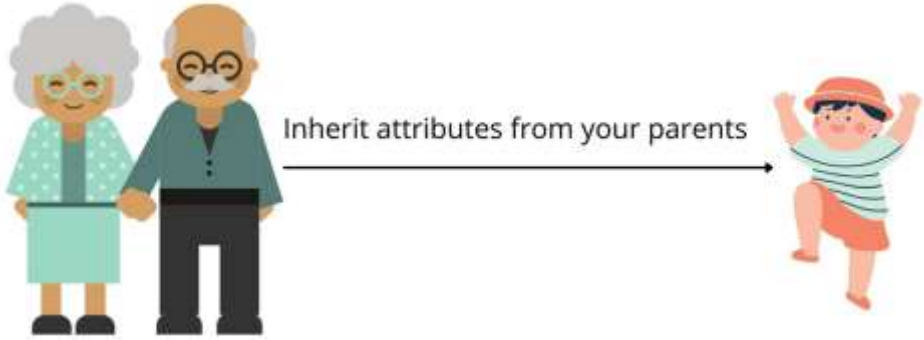
- **Base class** (or parent or superclass) – inherited from
- **Derived class** (or child or subclass) – inherits from the base class
- **Notation:**

```
class Student                    // base class
{

};

class UnderGrad : public Student
{                                              // derived
class

};
```

Inheriting Data and Functions

- All data members and member functions of base class are inherited to derived class, except

- Constructors, destructors and = operator are ***not inherited***



What Does a Child Class Have?

In the Student and underGrad example shown earlier:

An object of the *derived class* has:

- all members defined in child class
- all members of the parent class except constructors, destructors and operator=

An object of the *derived class* can use:

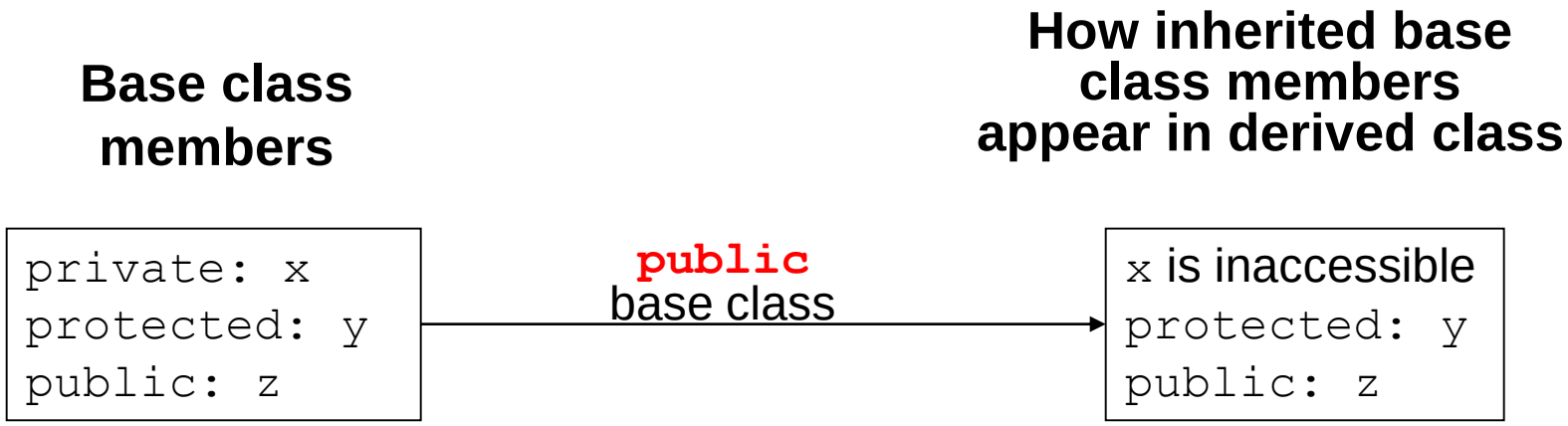
- all **`public`** members defined in child class
- all **`public`** members defined in parent class

```
//*********Inheritance Example***********
//Car is "IS-A" Vehicle
class Vehicle {
private:
    int speed;
public:
    Vehicle() {
        speed = 0;
        cout << "\nVehicle constructor" <<
endl;
    }
    void setSpeed(int spd) {
        speed = spd;
    }
    int getSpeed() {
        return speed;
    }
    void start() {
        cout << "\nStart Vehicle" << endl;
    }
    void stop() {
        cout << "\nStop Vehicle" << endl;
    }
};
```

```
class Car : public Vehicle {
private:
    int wheels;
public:
    Car() {
        wheels = 0;
        cout << "\nCar constructor" <<
endl;
    }
    void accelerate() {
       // speed++;   cannot be
accessed directly, private
       //use setters and getters of
base class to access private members
        cout << "\nCar accelerating"
<< endl;
    }
};


int main()
{
    Car c1;
    c1.start();
    c1.accelerate();
    c1.stop();
    return 0;
}
```

Protected Members and Class Access

- **`protected`** member access specification: like **`private`**, but accessible by objects of derived class



```
//*********Inheritance Example***********
//Car is "IS-A" Vehicle
class Vehicle {
protected:
    int speed;
public:
    Vehicle() {
        speed = 0;
        cout << "\nVehicle constructor" <<
endl;
    }
    void setSpeed(int spd) {
        speed = spd;
    }
    int getSpeed() {
        return speed;
    }
    void start() {
        cout << "\nStart Vehicle" << endl;
    }
    void stop() {
        cout << "\nStop Vehicle" << endl;
    }
};
```

```
class Car : public Vehicle {
private:
    int wheels;
public:
    Car() {
        wheels = 0;
        cout << "\nCar constructor" <<
endl;
    }
    void accelerate() {

          speed++;   //works,
protected

cout << "\nCar accelerating" << endl;
    }
};


int main()
{
    Car c1;
    c1.start();
    c1.accelerate();
    c1.stop();
    return 0;
}
```

Protected Members and Class Access

- **`protected`** member access specification: like **`private`**, but accessible by objects of derived class

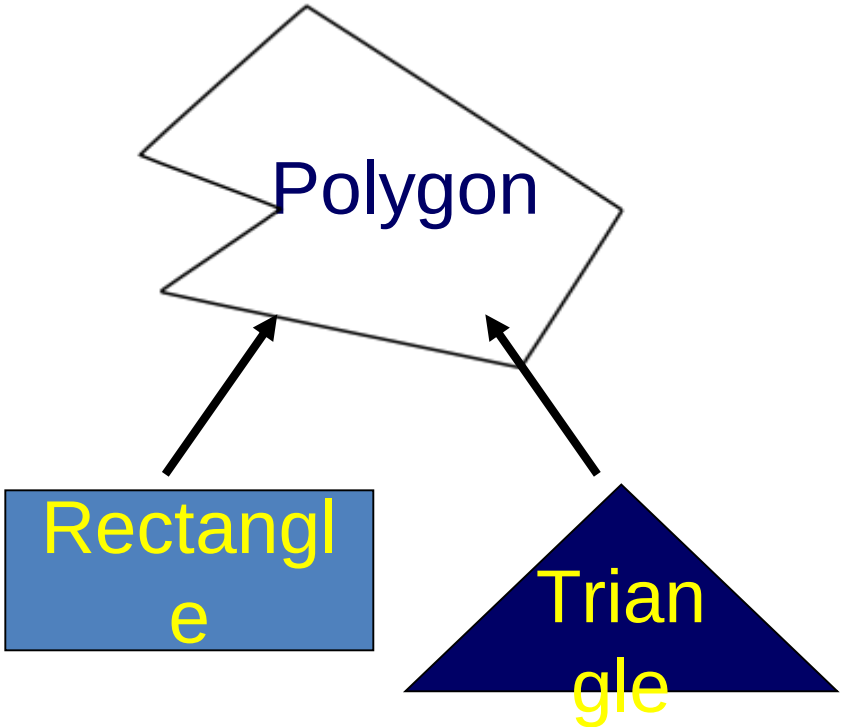
Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

Protected Members and Class Access

- **`protected`** member access specification: like **`private`**, but accessible by objects of derived class

- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Inheritance Example 1

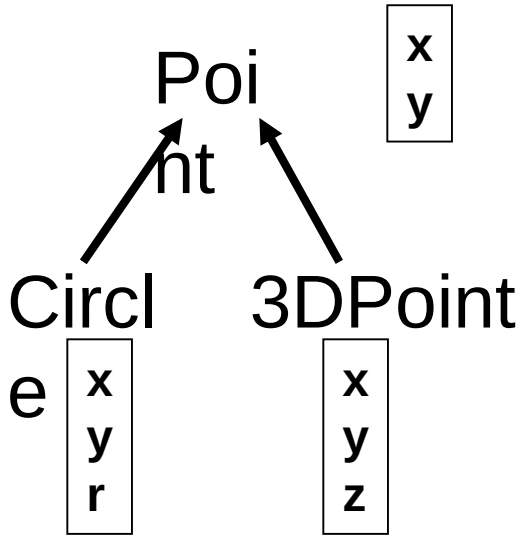



```
class Polygon{
protected:
   int numVertices;
   float *xCoord, float *yCoord;
public:
void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
   public:
      float area();
};
```

```
class Triangle : public Polygon{
public:
   float area();
};
```

Inheritance Example 2



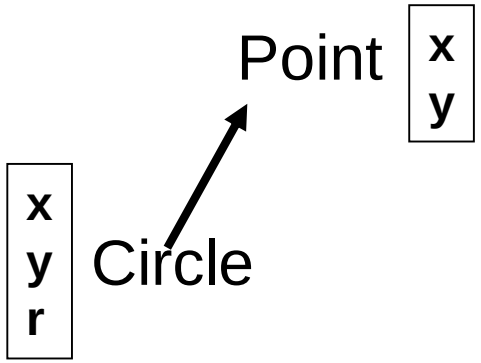

```
class Point{
protected:
   int x, y;
public:
   void set (int a, int b);
};
```

```
class Circle : public Point{
private:
   double r;
};
```

```
class 3DPoint: public Point{
private:
    int z;
};
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class




```
class Circle : public Point{
 private:
   double r;
 public:
   void set_r(double c);
};
```

```
class Point{
protected:
  int x, y;
public:
  void set(int a, int b);
};
```

```
class Circle{
protected:
   int x, y;
private:
   double r;
public:
   void set(int a, int b);
   void set_r(double c);
};
```

Dangers of Protected

- You should know that there's a disadvantage to making class members protected.
- Say you've written a class library, which you're distributing to the public. Any programmer can access protected members of your classes simply by deriving other classes from them.
- This makes protected members considerably less secure than private members.
- To avoid corrupted data, it's often safer to force derived classes to access private data in the base class using only public setters and getters.

Constructors and Destructors in Base and Derived Classes

- Constructors and destructors of Base class are ***NOT inherited***
- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructor Rules for Derived Classes

- The default constructor and the destructor of the base class are ***always called*** when a new object of a derived class is created or destroyed.

```
class A {
public:
   A ( )
     { cout<< "A default"<<endl; }
   A (int a)
     { cout<<"A parametrized"<<endl;
   }
};
```

```
class B : public A
{
  public:
   B ( ) {
   cout<<"B default"<<endl; }
};
```

```
B obj;
```

output:

```
A default
B default
```

Constructor Rules for Derived Classes

- The default constructor and the destructor of the base class are ***always called*** when a new object of a derived class is created or destroyed.

```
class A {
public:
    A ( )
      { cout<< "A default"<<endl; }
    A (int a)
      { cout<<"A parametrized"<<endl;
    }
};
```

```
class B : public A
{
  public:
    B (int a) {
    cout<<"B parametrized"<<endl; }
};
```

```
B obj(1);
```

output:

```
A default
B parametrized
```

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:
- Inline constructor syntax:

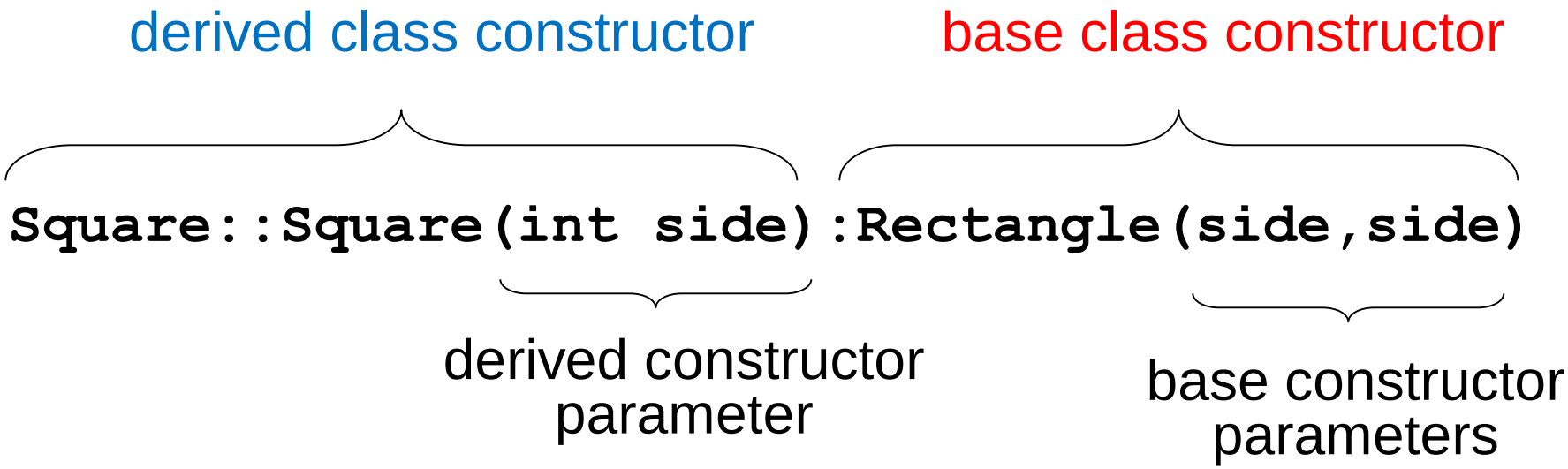
```
Square(int side) : Rectangle(side, side)
```

- Can also be done with out-of-line constructors

```
Square::Square(int side):Rectangle(side, side)
```

- Must be done if ***base class has no default constructor***

Passing Arguments to Base Class Constructor



Constructor Rules for Derived Classes

- You can specifically call a constructor of the base class other than the default constructor

```
DerivedClassCon ( derivedClass args ) : BaseClassCon ( baseClass args )

    {  DerivedClass constructor body   }
```

```
class A {
   public:
   A ( )
     { cout<< "A default"<<endl; }
   A (int a)
     { cout<<"A parametrized"<<endl;
   }
};
```

```
class C : public A {
   public:
   C (int a) : A(a) {

   cout<<"C parametrized"<<endl; }
};
```

C test(1);

output :

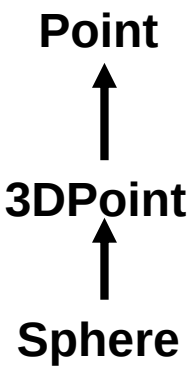
A parametrized
C parametrized

```
//*********Inheritance Example************
//Square is "IS-A" Rectangle (with l = w)
class Rectangle {
protected:
    int length;
    int width;
public:
    Rectangle() {
        length = 0; width = 0;
        cout << "\nRectangle default" << endl;
        cout << "Length " << length << endl;
        cout << "Width " << width << endl;
    }

    Rectangle(int l, int w) {
        length = l; width = w;
        cout << "\nRectangle parametrized" << endl;
        cout << "Length " << length << endl;
        cout << "Width " << width << endl;
    }

    ~Rectangle() {
        cout << "\nRectangle destructor" << endl;
    }
};
```

```
class Square : public Rectangle{
private:
    int side;
public:
    Square() {
        side = 0;
        cout << "\nSquare default" << endl;
        cout << "Side " << side << endl;
    }

    Square(int s) : Rectangle(s,s) {
        side = s;
        cout << "\nSquare parametrized" << endl;
        cout << "Side " << side << endl;
    }

    ~Square() {
        cout << "\nSquare destructor" << endl;
    }
};
int main()
{
    Square obj(5);
    return 0;
}
```

```
Rectangle parametrized
Length 5
Width 5

Square parametrized
Side 5

Square destructor

Rectangle destructor
```

Class Derivation




```
class Point{
protected:
   int x, y;
public:
   void set (int a, int b);
};
```

```
class 3DPoint : public Point {
   private:
         double z;

};
```

```
class Sphere : public 3DPoint{
private:
   double r;

};
```

Point is the **base class** of **3D-Point**, while **3DPoint** is the **base class** of **Sphere**

Order of execution of Constructors/Destructors

- Chain of constructor calls:

`Point` → `Circle` → `Cylinder`

  - Point constructor executes first
  - Then circle and last Cylinder

Order of execution of Constructors/Destructors

- Chain of destructor calls
  - Reverse order of constructor chain

```
Cylinder □  Circle □  Point
```

  - Destructor of derived-class called first
  - Destructor of next base class up hierarchy next
  - Continue up hierarchy until final base reached

  - After final base-class destructor, object removed from memory

Example

```
class Point
{
        protected:
                        int x, y;
        public:
                        Point(int ,int );
                        void display(void);
                        ~Point() { cout<<"\nPoint Class
Destructor\n"; }
};

Point::Point(int a,int b) {
        cout<< "\nPoint Class Constructor\n";
        x = a;
        y = b;
}

void Point::display(void) {
        cout<< "point = [" << x <<","<< y <<"]";
}
```

Example – cont.

```
class Circle : public Point
{
        protected:
                double radius;
        public:
                Circle(int ,int ,double);
                void display(void);
                ~Circle() { cout <<"\n Circle Class Destructor
\n"; }
};

Circle::Circle(int a,int b,double c):Point(a,b) {
        cout <<"\n Circle Class Constructor "<<endl;
        radius = c;
}

void Circle::display(void) {
        Point::display(); //Parent class display called
        cout <<" radius = " << radius;
}
```

Example – cont.

```
class Cylinder: public Circle
{
        double height;
        public:
                Cylinder(int ,int ,double ,double);
                void display(void);
                double GetVolume(void);
                ~Cylinder() { cout<<"\nCylinder Class
Destructor\n"; }
};

Cylinder::Cylinder(int a,int b,double r,double h):Circle(a,b,r)
{
        cout << "\nCylinder Class Constructor"<<endl;
        height=h;
}

double Cylinder::GetVolume(void) {
        return 3.14 * radius * radius * radius;
}
```

Example – cont.

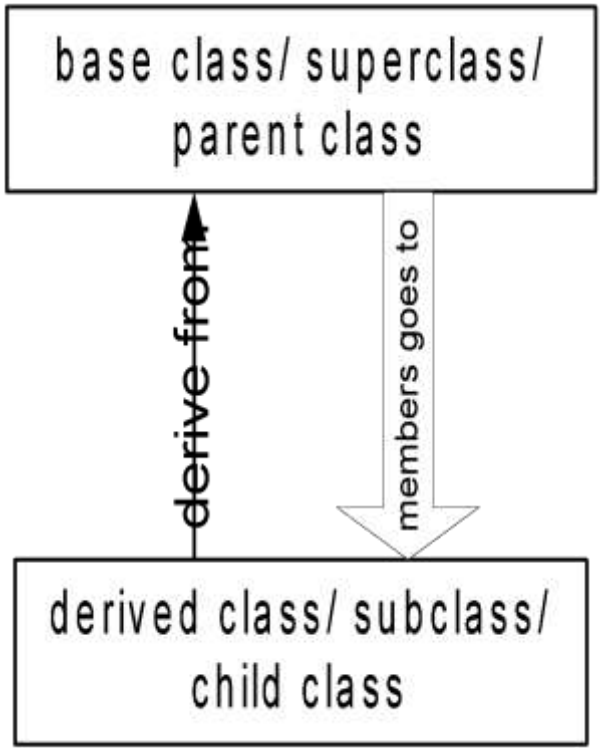
```
int main(void)
{
      Cylinder c(3, 4, 2.5, 3.7);
      return 0;
}
```

Output:

Point Class Constructor
Circle Class Constructor
Cylinder Class Constructor
Cylinder Class Destructor
Circle Class Destructor
Point Class Destructor

Data vs Class Access Specifier

- Two levels of access control over class members
  1. class definition
  2. inheritance type

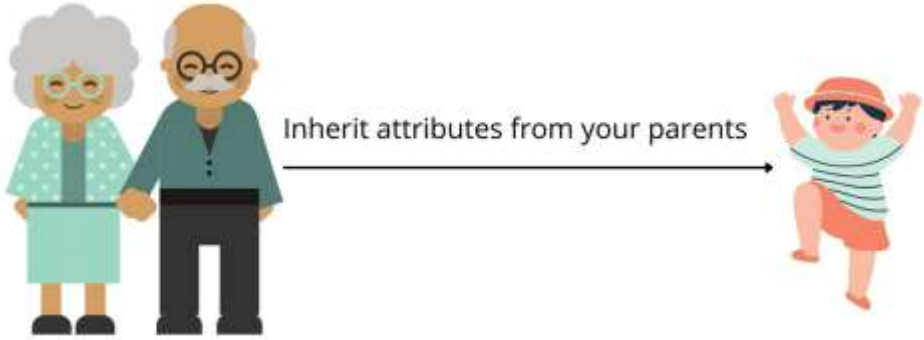



```
class Point{
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point{
    … …
};
```

Types of inheritance/ Class access specifier

- public
- private
- protected



Public Inheritance

- With public inheritance,
  - public and protected members of the base class become respectively public and protected members of the derived class

```
class derived : public base{
        … …
};
```

Protected Inheritance

- Public and protected members of the base class become ***protected members*** of the derived class.

```
class derived : protected base{
        … …
};
```

Private Inheritance

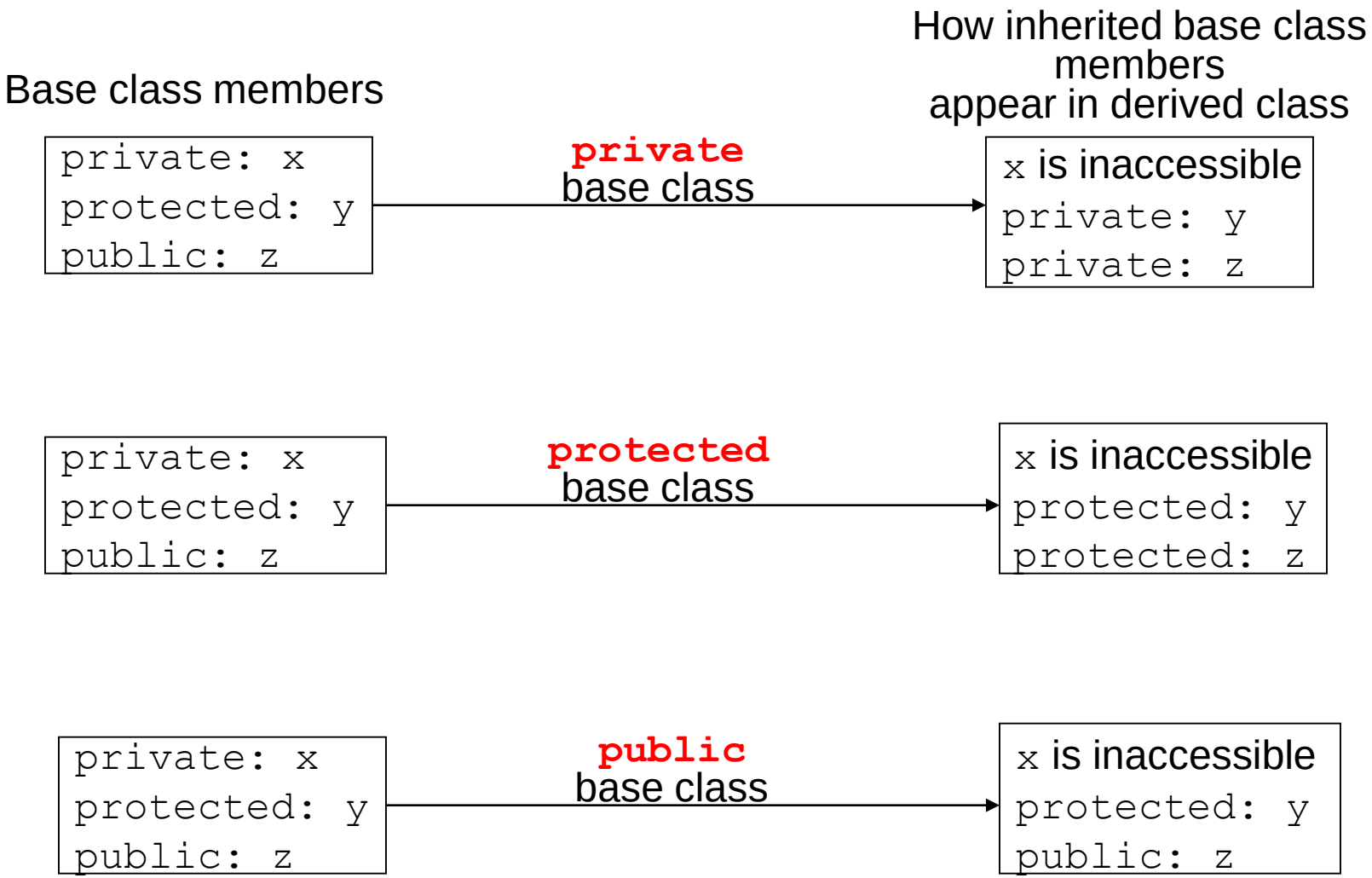
- With private inheritance, public and protected members of the base class become ***private members of the derived class***.

```
class derived : private base{
        … …
};
```

public, protected and private Inheritance

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Inheritance vs. Access



Class Access Specifiers – When to use?

1) **`public`** – object of derived class can be treated as object of base class (not vice-versa)

1) **`protected`** – more restrictive than **`public`**, but allows derived classes to know details of parents

1) **`private`** – prevents objects of derived class from being treated as objects of base class.

Method Overriding

- A derived class can override methods defined in its parent class.
  - the method in the subclass must have the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
protected:
   int x, y;
public:
   void print ()
   { cout<<"From A"<<endl; }
};
```

```
class B : public A {
   public:
   void print ()
        {cout<<"From B"<<endl;}
};
```

Method Overriding

```
class Point{
protected:
   int x, y;
public:
 void set(int a, int b)
     { x=a; y=b; }
 void foo ();
 void print();
};
```

```
class Circle : public Point{
private:  double r;
public:
//function overriding
//In inheritance base class functions
   are not overloaded. They are
   overridden.
 void set (int a, int b, double c) {
  Point :: set(a, b); //same name function call
  r = c;
  }
//function overriding
  void print();
};
```

Point A;

A.set(30,50); // from base class Point

A.print(); // from base class Point

Circle C;

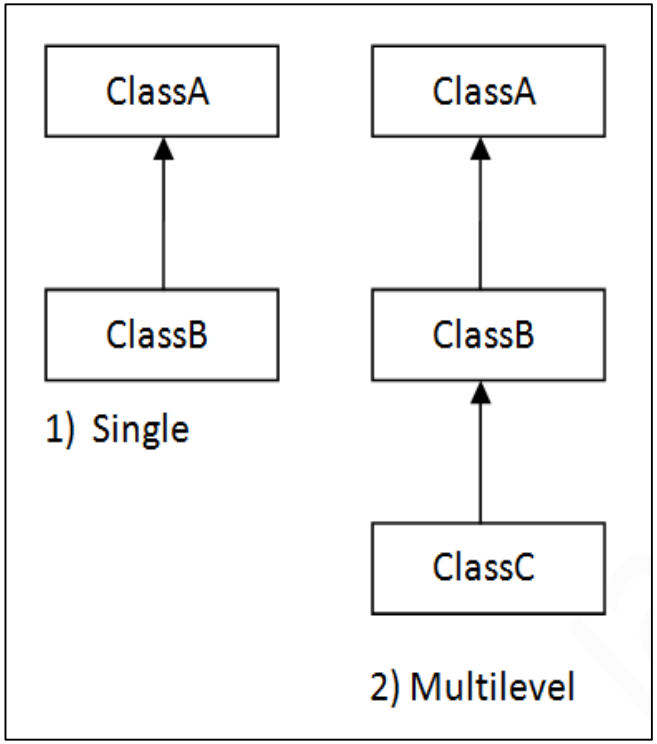
C.set(10,10,100); // from class Circle

C.foo (); // from base class Point

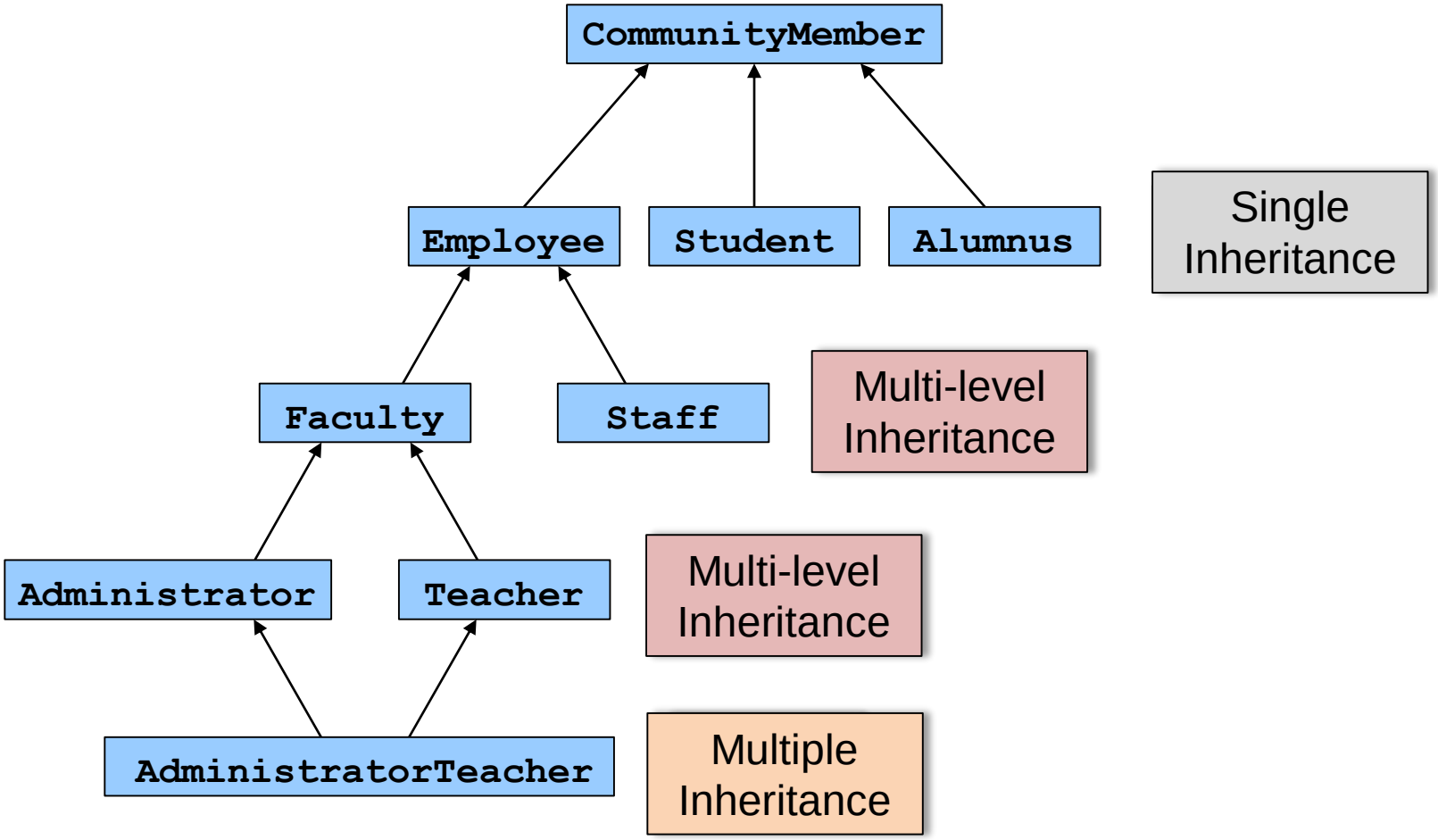
C.print(); // from class Circle

Types of Inheritance

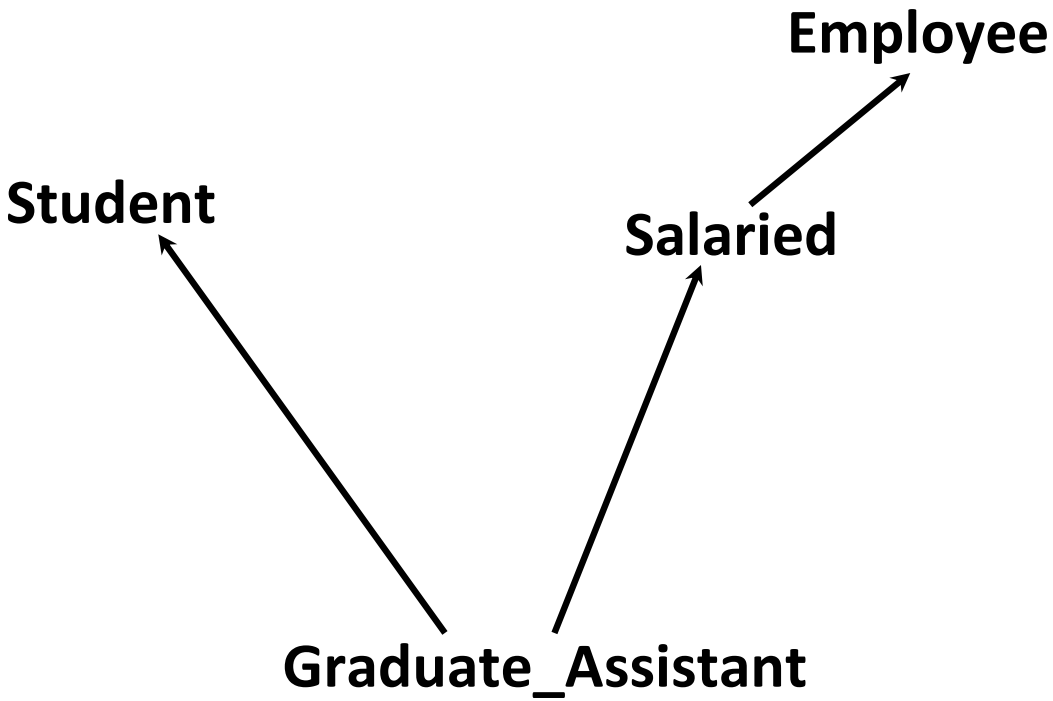
- **Single inheritance**
  - Inherits from one base class
- **Multi-level inheritance**
  - Chain of inheritance
- **Multiple inheritance**
  - Inherits from multiple classes



Types of Inheritance: University Example



Multiple Inheritance



What is Multiple Inheritance?

- If class A ***inherits from more than one class***,
  - i.e., A □ (B1, B2, ..., Bn), we speak of multiple inheritance.

- This may introduce naming conflicts:
  - if at least two of its base classes define properties (data members or member functions) with the same name

```
//simple example showing multiple inheritance
class A {
public:
        void fun1(void)  { cout<<"fun1";       }
};


class B {
public:
        void fun2(void)         { cout<<"fun2"; }
};


class derived: public A, public B {
public:
        void funderived(void)  { cout<<"func derived";}
};



void main(void) {
        derived der;
        der.fun1();
        der.fun2();
        der.funderived();
}
```

Ambiguity in Multiple Inheritance

```
class Student {
      int id;
      int age;

public:
      int GetAge() const { return age; }
      int GetId() const { return id; }
      void SetAge( int n ) { age = n; }
      void SetId( int n ) { id=n; }
};
```

Ambiguity in Multiple Inheritance

```
class Employee {

public:
      int GetAge() const { return age; }
      int SetAge( int n ) { age = n; }
      void SetId( int n) { id=n; }
      int GetId(void) const { return id; }

private:
      int age;
      int id;
};
```

Ambiguity in Multiple Inheritance

```
class Salaried : public Employee {
        float salary;
public:
        float GetSalary() const { return salary; }
        void SetSalary( float s ) { salary=s; }
};

class GradAssistant :public Student, public Salaried {
public:
  void Display() const
{
   cout<<GetId()<<","<<GetSalary()<<","<<GetAge(); //ambiguity
}
```

Ambiguity in Multiple Inheritance

```
int main(void) {

 GradAssistant ga;
 ga.SetAge(20);  //ambiguity
 ga.SetId(15);   //ambiguity
 ga.Display();   //ambiguity inside display()

}
//program will not compile and will generate
errors
```

What is the solution?

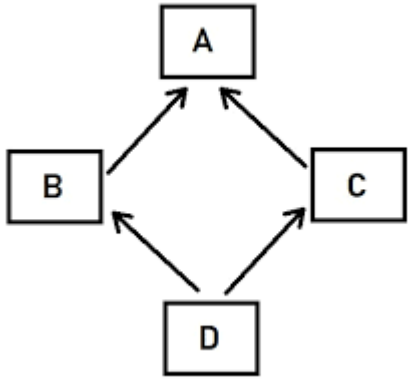
- Call functions explicitly by specifying name of class and using scope resolution operator to remove ambiguity:

1. Direct solution:

```
Student::SetAge() or
Salaried::SetAge()
```

The Diamond Problem

```
class A {
      public:
            void Foo() {}
}
class B : public A {}
class C : public A {}
class D : public B, public C {}
```




```
D d;
d.Foo();
```

is this B's Foo() or C's Foo() ?? ***ambiguous***

What is the solution?

- Call functions explicitly by specifying name of class and using scope resolution operator to remove ambiguity:

1. Direct solution:

```
Student::SetAge() or
Salaried::SetAge()
```

2. Virtual inheritance

Solution (virtual inheritance)

```
class A {
       public: void Foo() {}
}

class B : public virtual A {
}

class C : public virtual A {
}

class D : public B, public C {
}


D d;
d.Foo(); // no longer ambiguous
```

Virtual Inheritance

Virtual inheritance is a C++ technique that ensures that ***only one copy of common base class's member variables are inherited by second-level derivatives*** (a.k.a. grandchild derived classes)

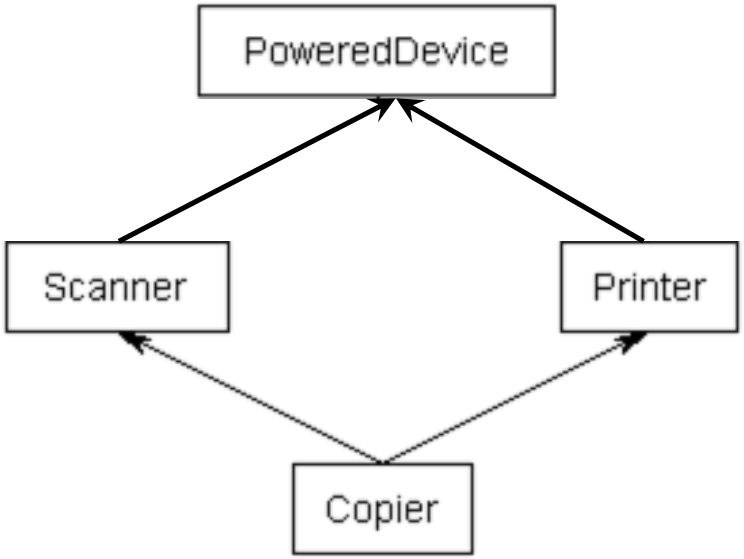
Another example

```
class PoweredDevice
{
      public:
      PoweredDevice(int nPower)
        {
                        cout << "PoweredDevice:
" <<
                        nPower << endl;
        }
};
```

```
class Scanner: public PoweredDevice {
public:
  Scanner(int nScanner, int nPower) : PoweredDevice(nPower)
    {
        cout << "Scanner: " << nScanner << endl;
    }
};

class Printer: public PoweredDevice {
public:
  Printer(int nPrinter, int nPower) : PoweredDevice(nPower)
    {
        cout << "Printer: " << nPrinter << endl;
    }
};
```

Another example

```
class Copier: public Scanner, public Printer
{
public:
Copier(int nScanner, int nPrinter, int nPower) :
Scanner(nScanner, nPower), Printer(nPrinter, nPower)
    {
    }
};
```



Another example

```
int main()
{
    Copier cCopier(1, 2, 3);
}
```

What should be the output?

PoweredDevice: 3

Scanner: 1

PoweredDevice: 3

Printer: 2