Advanced Programming

Inheritance Relationship

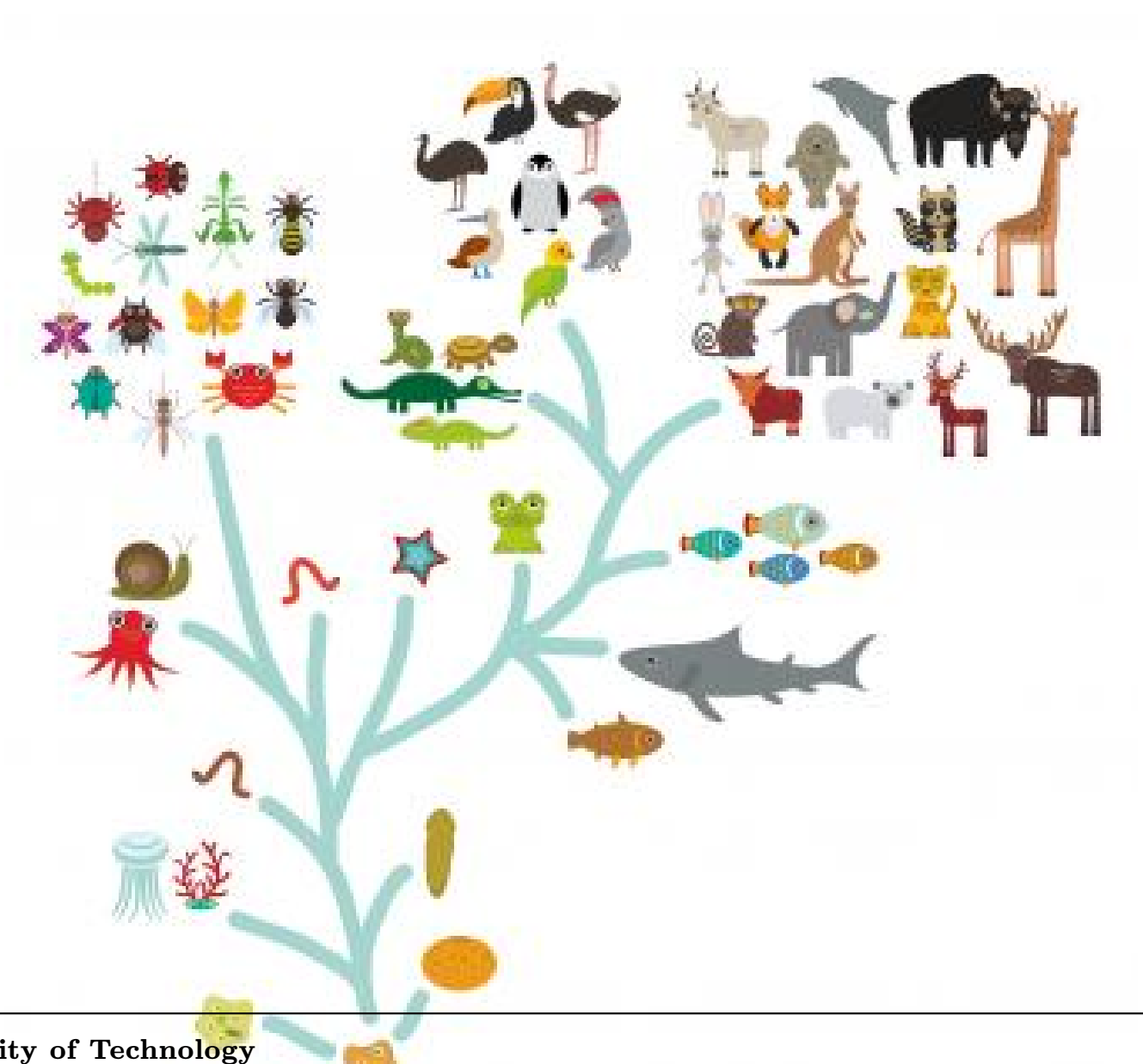
Zeinab Zali

References: (1) "C++ How to program" Deitel&Deitel, (2) "A Tour of C++" Bjarne Stroustrup, (3) Other useful learning pages such as geeksforgeeks and tutorialpoints

ECE Department, Isfahan University of Technology

Lions are cats

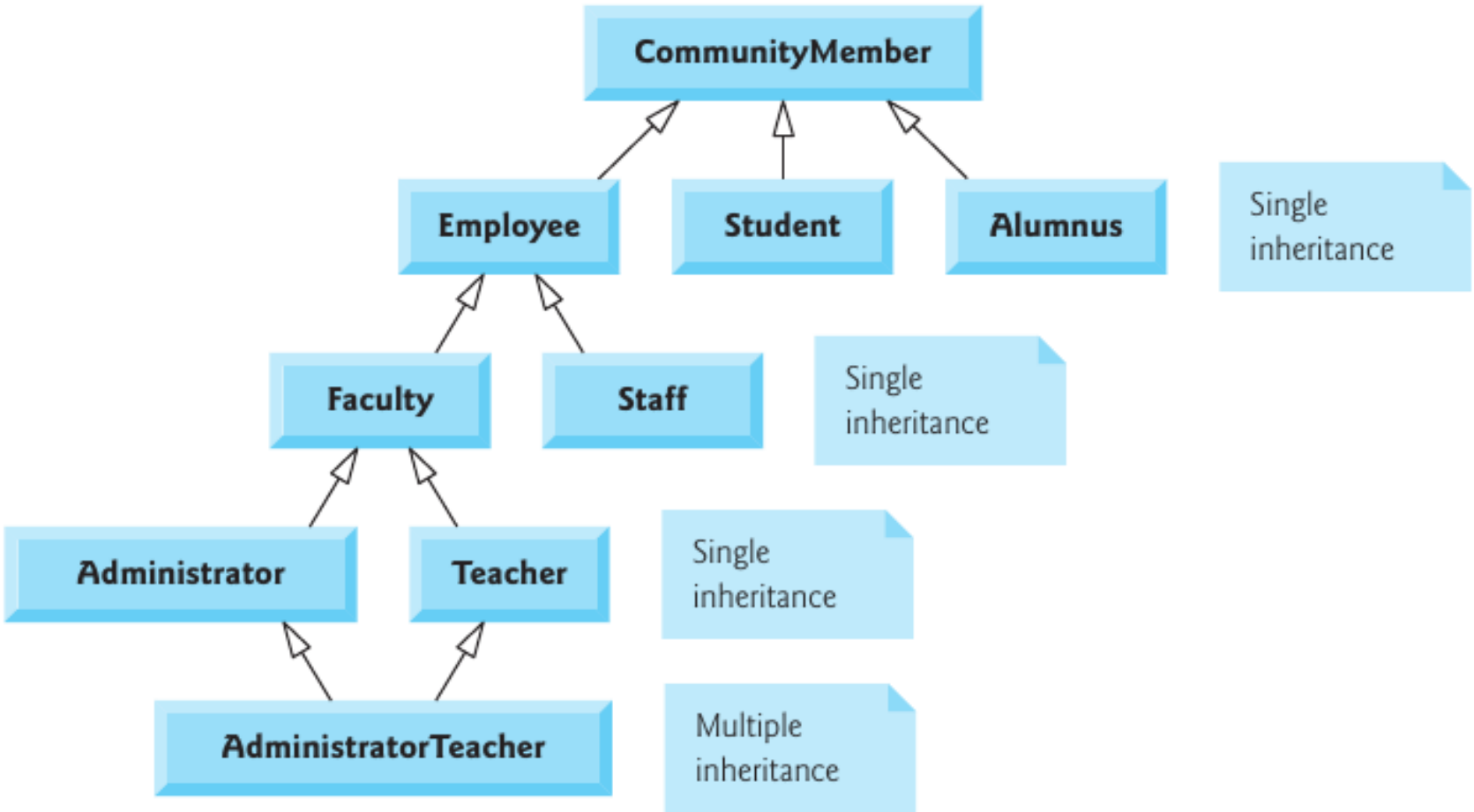
Inheritance and Evolution



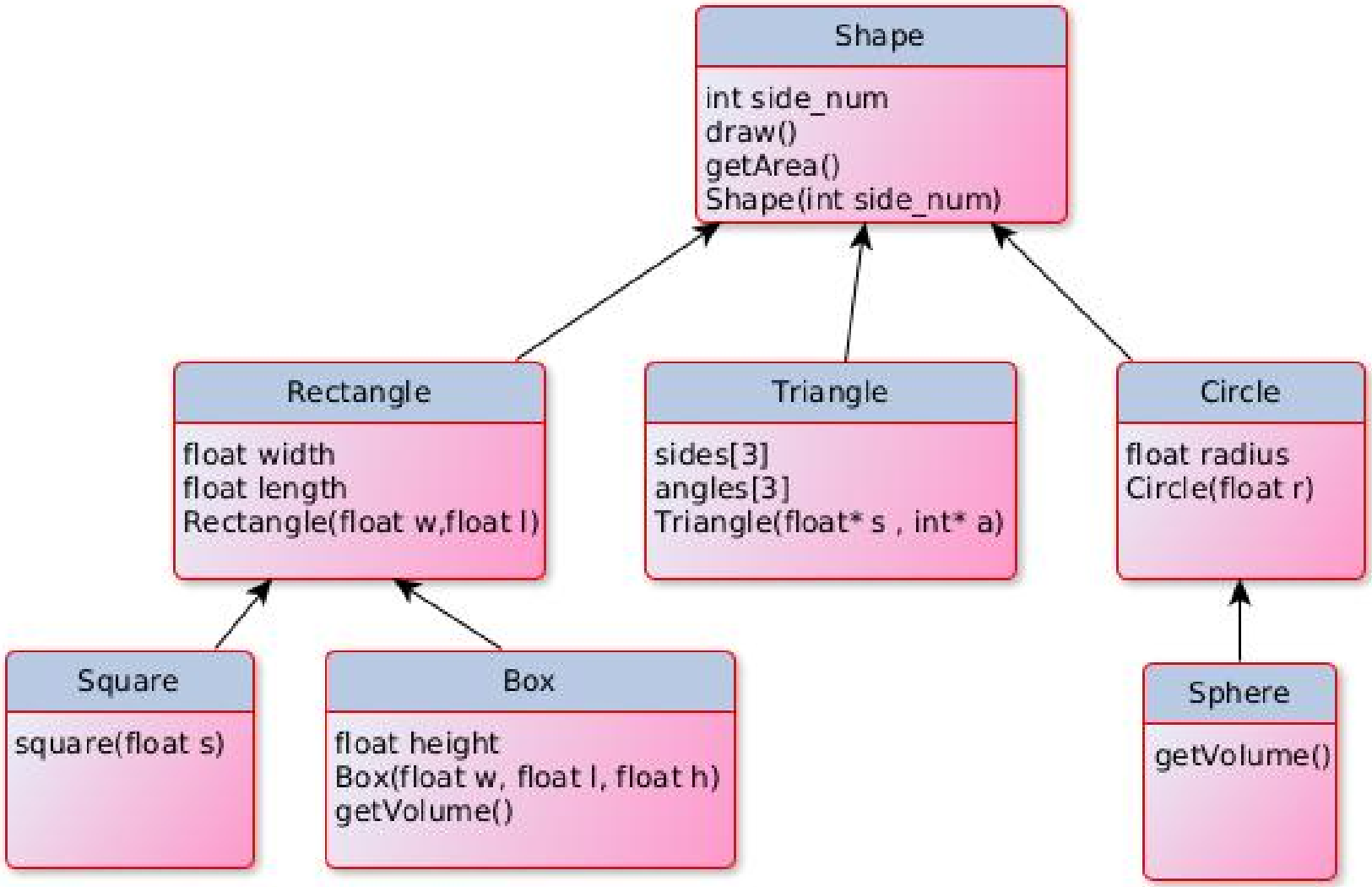
Inheritance in Objects

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

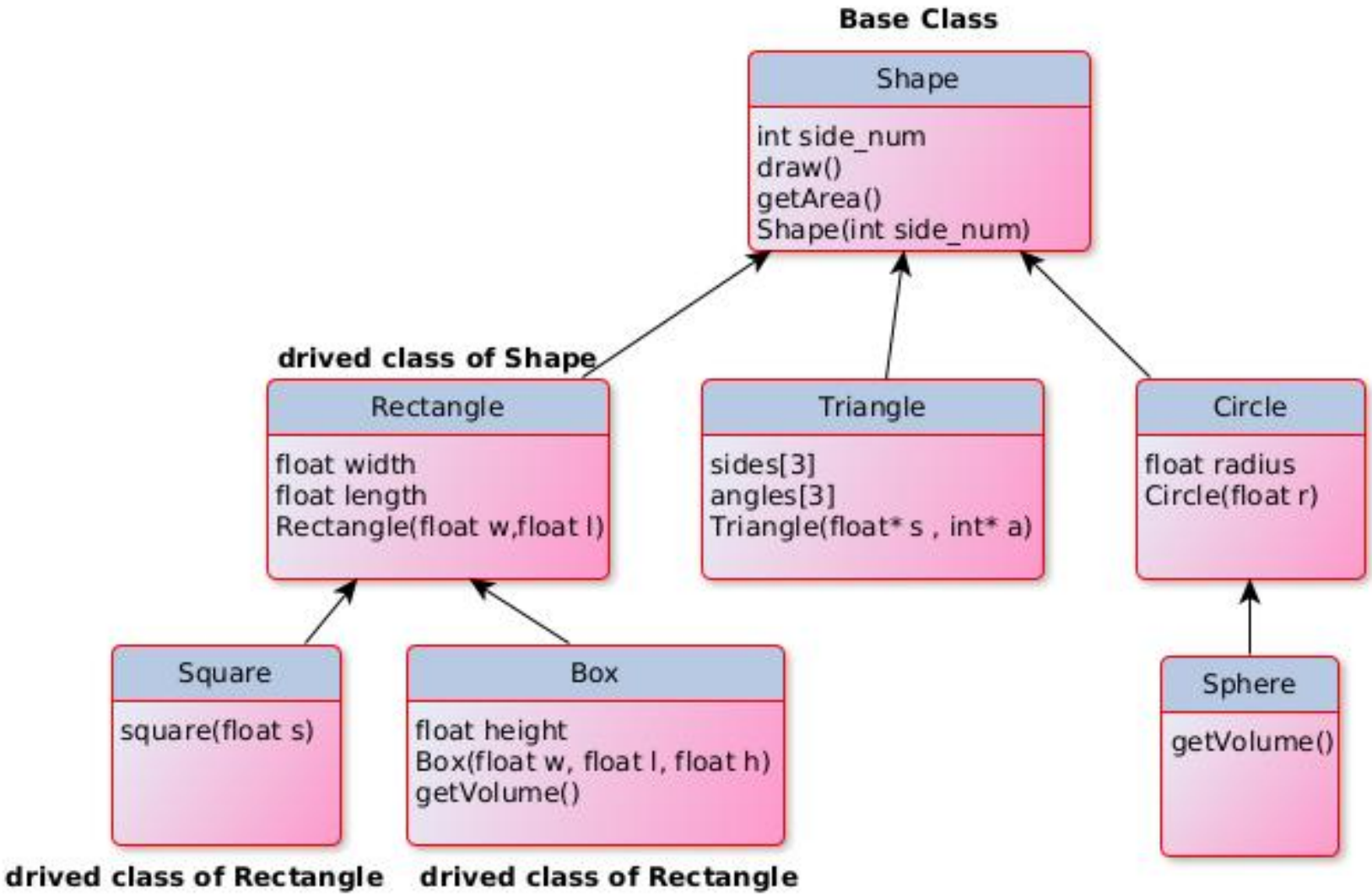
Inheritance in Objects



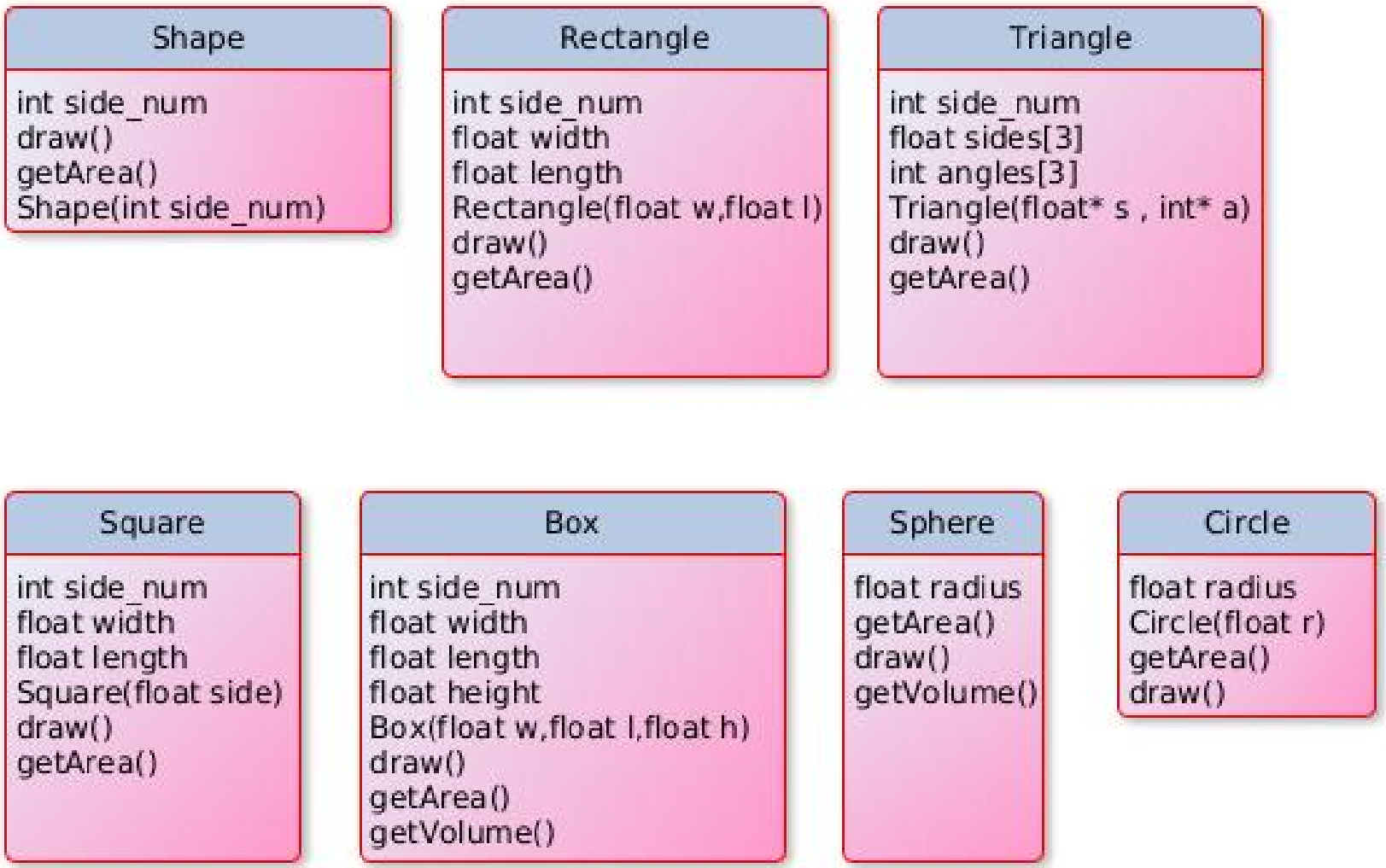
Inheritance in Objects



Class Hierarchies



Without using Inheritance



Without using Inheritance

Most of the codes for these classes are similar.

Software Engineering Observation 11.1

Copying and pasting code from one class to another can spread many physical copies of the same code and can spread errors throughout a system, creating a code-maintenance nightmare. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.

Software Engineering Observation 11.2

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only *in the base class—derived classes then inherit the changes. Without inheritance, changes would need to be made to* all *the source-code files that contain a copy of the code in question.*

Base and Drived Classes

Inheritance enables you to create a class that absorbs an existing class's capabilities, then customizes or enhances them.

- The existing class is called the **base class** (super class), and the new class is called the **derived class** (sub class).
- A derived class represents a more specialized group of objects.

is-a relationship

- With **public inheritance**, every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes.
- For example, if we have Vehicle as a base class and Car as a derived class, then all Cars are Vehicles, but not all Vehicles are Cars—for example, a Vehicle could also be a Truck or a Boat.

is-a relationship

- The **is-a relationship** represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class—for example, a Car is a Vehicle, so any attributes and behaviors of a Vehicle are also attributes and behaviors of a Car.
- By contrast, the **has-a relationship** represents composition. In a has-a relationship, an object contains one or more objects of other classes as members. For example, a Car has many components—it has a steering wheel, has a brake pedal, has a transmission

Base Class: Shape

```
1 class Shape{
2         const int sidesNum;
3     public:
4         Shape():sidesNum(4){
5             cout << "Shape default Constructor\n";
6         }
7         Shape(int s):sidesNum(s){
8             cout << "Shape Constructor\n";
9         }
10        int getSidesNum(){
11            return sidesNum;
12        }
13        int getArea(){
14            cout << "Shape getArea()\n";
15            return 1;
16        }
17        ~Shape(){
18            cout << "Shape destructor\n";
19        }
20 };
```

Derived Class: Rectangle

```
1class Rectangle: public Shape{
2         float len;
3         float width;
4     public:
5         Rectangle(){
6              cout << "Rectangle default Constructor\n";
7              len = width =1;
8              // sidesNum = 4; //error, sidesNum is
    inaccessible
9         }
10        Rectangle(float l, float w):Shape(4){
11             cout << "Rectangle Constructor\n";
12             len = l;
13             width = w;
14        }
15        int getArea(){
16             cout << "Rectangle getArea\n";
17             return len*width;
18        }
19        ~Rectangle(){
20             cout << "Rectangle destructor\n";
21        }
```

Introducing Protected Access Specifier

- A base class's **public** members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of that base class's derived classes.
- A base class's **private** members are accessible only within the base class or from its friends.
- A base class's **protected** members can be accessed by members and friends of that base class and by members and friends of any classes derived from that base class.

Shape with protected data member

```
1 class Shape{
2     protected:
3         const int sidesNum;
4     public:
5         Shape():sidesNum(0){
6             cout << "Shape default Constructor\n";
7         }
8         Shape(int s):sidesNum(s){
9             cout << "Shape Constructor\n";
10        }
11        int getSidesNum(){
12            return sidesNum;
13        }
14        float getArea(){
15            cout << "Shape getArea()\n";
16            return 0;
17        }
18        ~Shape(){
19            cout << "Shape destructor\n";
20        }
21 };
```

Notes on protected

Using protected data members creates two serious problems.

- The derived-class object does not have to use a member function to set the value of the base class's protected data member. An invalid value can easily be assigned to the protected data member, thus leaving the object in an inconsistent state
- Derived classes should depend only on the base-class services (i.e., non- private member functions) and not on the base-class implementation. With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class.

Inherited Members

Derived class inherits all the members of base class, except for the constructor—each class provides its own constructors that are specific to the class. (Destructors, too, are not inherited.)

Constructors and Destructors in Derived Classes

- When an object of a derived class is instantiated, the base class's constructor is called immediately to initialize the base-class data members in the derived-class object, then the derived-class constructor initializes the additional derived-class data members.
- When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors—first the derived-class destructor is called, then the base-class destructor is called.

Constructors and Destructors in Derived Classes

- For each derived class, C++ attempts to invoke base class's default constructor implicitly
- If there is not a default constructor in the base class, we must explicitly call one of the constructors of the base class in member initializer list.

Base Class with a Default Constructor

```
1          Shape():sidesNum(0){
2               cout << "Shape default Constructor\n";
3          }
```

```
1          Rectangle(){
2               cout << "Rectangle default Constructor\n";
3               len = width =1;
4          }
```

```
1     int main(){
2          Rectangle r1;
3          cout << "r1 area: \n"<< r1.getArea() <<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape default Constructor
Rectangle default Constructor
r1 area:
Rectangle getArea
1
Rectangle destructor
Shape destructor
```

Base Class with a Default Constructor

```
1          Shape():sidesNum(0){
2              cout << "Shape default Constructor\n";
3          }
4          Shape(int s):sidesNum(s){
5              cout << "Shape Constructor\n";
6          }
```

```
1          Rectangle(float l, float w){
2              cout << "Rectangle Constructor\n";
3              len = l;
4              width = w;
5          }
```

```
1      int main(){
2          Rectangle r2{2,3};
3          cout << "r2 area: "<< r2.getArea() <<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape default Constructor
Rectangle Constructor
r2 area: Rectangle getArea
6
Rectangle destructor
Shape destructor
```

Base Class without a Default Constructor

```
1          Shape(int s):sidesNum(s){
2               cout << "Shape Constructor\n";
3          }
```

```
1          //error, no default constructor for shape
2          Rectangle(){
3               cout << "Rectangle default Constructor\n";
4               len = width =1;
5          }
6          //error, no default constructor for shape
7          Rectangle(float l, float w){
8               cout << "Rectangle Constructor\n";
9               len = l;
10              width = w;
11         }
```

```
Shape4.cpp: In constructor 'Rectangle::Rectangle()':
Shape4.cpp:30:20: error: no matching function for call to 'Shape::Shape()'
   30 |         Rectangle(){
      |                    ^
Shape4.cpp:10:9: note: candidate: 'Shape::Shape(int)'
   10 |         Shape(int s):sidesNum(s){
      |         ^~~~~
```

Explicitly Calling a Constructor in Member-initializer list

```
1   Shape(int s):sidesNum(s){
2       cout << "Shape Constructor\n";
3   }
```

```
1   Rectangle():Shape(4){
2       cout << "Rectangle default Constructor\n";
3       len = width =1;
4   }
```

```
6   Rectangle(float l, float w):Shape(4){
7       cout << "Rectangle Constructor\n";
8       len = l;
9       width = w;
10  }
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape Constructor
Rectangle Constructor
r2 area: Rectangle getArea
6
Rectangle destructor
Shape destructor
```

Overriding

- If the derived class inherit a method from the base class, calling that method for the objects of derived class, results in executing the base class method
- We can redefine a method of base class in a derived class (**overriding**). In this situation, calling that method for the objects of derived class, results in executing the derived class method

Derived Class with overriden method

```
float Shape::getArea(){
    cout << "Shape getArea()\n";
    return 0;
}
```

```
int Rectangle::getArea(){
    cout << "Rectangle getArea\n";
    return len*width;
}
```

```
Rectangle r2{2,3};
cout << "r2 area: "<< r2.getArea() <<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape Constructor
Rectangle Constructor
r2 area: Rectangle getArea
6
Rectangle destructor
Shape destructor
```

Derived Class with Inherited Method of Base Class

```
1          float Shape::getArea(){
2              cout << "Shape getArea()\n";
3              return 0;
4          }
```

```
1          // float Rectangle::getArea(){
2          //     cout << "Rectangle getArea\n";
3          //     return len*width;
4          // }
```

```
1          Rectangle r2{2,3};
2          cout << "r2 area: "<< r2.getArea() <<endl;
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape Constructor
Rectangle Constructor
r2 area: Shape getArea()
0
Rectangle destructor
Shape destructor
```

Calling a Base Class Method in Derived Class

We can call base class methods explicitly in derived class, even if they are overridden in the derived class.

```
class Box: public Rectangle{
        float height;
    public:
        double getVolume(){
            cout << "Box getVolume()\n";
            return height * Rectangle::getArea();
        }
        float getArea(){
            cout << "Box getArea()\n";
            return 2 * (Rectangle::getArea() +
                    height * getWidth() +
                    height * getLen());
        }
```

Calling a Base Class Method in Derived Class

```
1   int main(){
2       Box b{10,20,30};
3       cout << fixed << setprecision(2) << b.getArea()<<
    endl;
4   }
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
Shape Constructor
Rectangle Constructor
Box Constructor
Box getArea()
Rectangle getArea()
2200.00
Box Destructor
Rectangle destructor
Shape destructor
```

Please pay attention to the hierarchical inheritanc too in this example!

C++11 allows you to specify that a derived class should inherit a base class's constructors.

- To do so, explicitly include a using declaration of the form using BaseClass::BaseClass;

```
1class Rectangle: public Shape{
2     //making all the constructors of Shape inherited in the
       Rectangle
3     using Shape::Shape;
```

Inheriting Constructors

```
1     public:
2          Rectangle():Shape(4){
3               cout << "Rectangle default Constructor\n";
4               len = width =1;
5          }

7          Rectangle(float l, float w):Shape(4){
8               cout << "Rectangle Constructor\n";
9               len = l;
10              width = w;
11         }
```

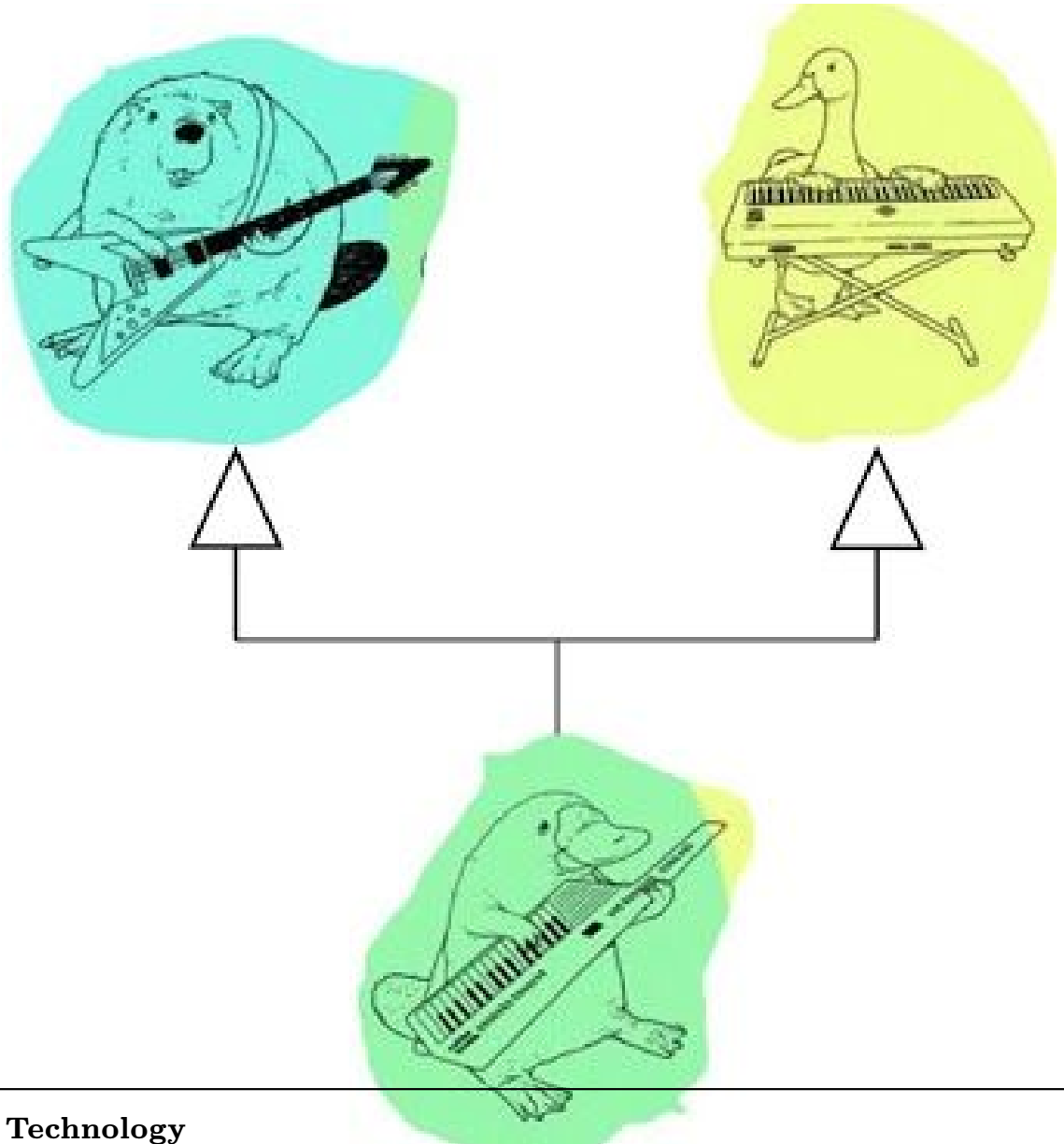
Inheriting Constructors

```
1   int main(){
2       cout << "creating r1:\n";
3       Rectangle r1(2); //creating r1 with the Shape
    constructor
4       cout << "creating r2:\n";
5       Rectangle r2;    //creating r2 with the Rectangle
    constructor, because default constructor is overriden
    in the Rectangle
6       cout << "\n";
7   }
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Shape
creating r1:
Shape Constructor
creating r2:
Shape Constructor
Rectangle default Constructor

Rectangle destructor
Shape destructor
Rectangle destructor
Shape destructor
```

Multiple Inheritance



Multiple Inheritance Example (Employee)

```
1 class Employee{
2         static long idGen;
3     protected:
4         long id;
5         long salary;
6     public:
7         Employee(long sal):salary(sal){
8             id = idGen++;
9             cout << "Employee Constructor\n";
10        }
11        ~Employee(){
12            cout << "Employee Destructor\n";
13        }
14        void print(){
15            cout << "Employee id: "<< id <<endl;
16        }
17 };
18 long Employee::idGen = 1;
```

Multiple Inheritance Example (Person)

```
1 class Person{
2     protected:
3         string name;
4         int age;
5     public:
6         Person(string s, int a):name(s), age(a){
7             cout << "Person Constructor\n";
8         }
9         ~Person(){
10            cout << "Person Destructor\n";
11        }
12        void print(){
13            cout << "Person name: "<< name<< endl;
14        }
15 };
```

Multiple Inheritance Example (Manager)

```
1class Manager: public Person, public Employee{
2     public:
3          Manager(string name, int age, long salary):Person(
     name,age), Employee(salary){
4               cout << "Manager Constructor\n";
5          }
6          ~Manager(){
7                    cout << "Manager Destructor\n";
8          }
9};
```

Ambiguity in Calling Member Functions

```
1 int main(){
2     Manager m1("Ali", 25, 30000);
3     //m1.print(); //error, Manager::print" is ambiguous
4     ((Employee)m1).print(); //resolving the ambiguous
5     m1.Person::print(); //resolving the ambiguous
6 }
```

```
~/Doc/IU/Te/AP/s/Ad/c/Inheritance ./Employee
Person Constructor
Employee Constructor
Manager Constructor
Employee id: 1
Employee Destructor
Person name: Ali
Manager Destructor
Employee Destructor
Person Destructor
```

public, protected and private Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.

- We normally use public inheritance.
- Use of protected inheritance is rare.

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.