Chapter 13

Inheritance and Polymorphism

Object Oriented Programming

The 3 principles of object oriented programming:

Encapsulation - the process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class.

Inheritance

Polymorphism

Inheritance

In object oriented programming, one of the most important topics is **inheritance.**

Inheritance - the concept of deriving a class from another class, forming a hierarchy.

Inheritance provides code reuse, better organization and faster development time.

Inheritance

When creating a class, instead of writing completely new data members and member functions, we can designate that the class inherit these properties from an existing class

Base Class (super class)- The existing class in inheritance. A class definition that provides basic, common data attributes and/or member functions that will be extended by a *derived* class.

Derived Class (sub class)- A class definition that inherits data members and member functions from a Base Class.

Derived Class Definition

To define a *derived class* we need to specify the access-specifier and the base class

```
class derived-class: access-specifier base-class
```

Types of access specifiers:

- Public
- Private
- Protected

example: base.cpp

UML Diagrams

One of the main principles of Object Oriented Programming is defining strong relationships between different pieces of information

We can represent these relationships using **UML Diagrams**.

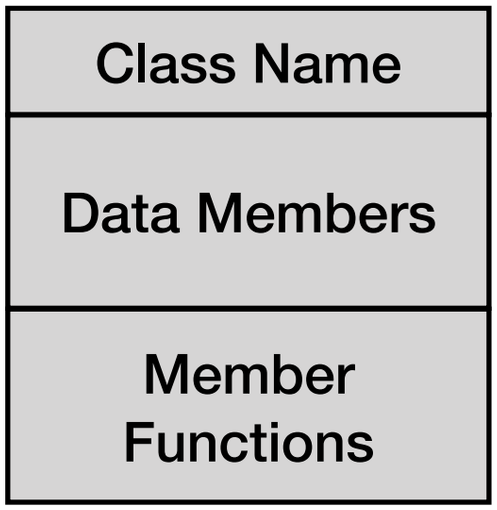
UML - Unified Modeling Language

UML is used in any Object Oriented Programming: Java, Python, C++, C#, etc.

UML Diagram Basics

A *Class Diagram* is a UML Diagram that depicts a class':

- Name
- Data Members (*public private and protected)*
- Member Functions (*public private and protected)*



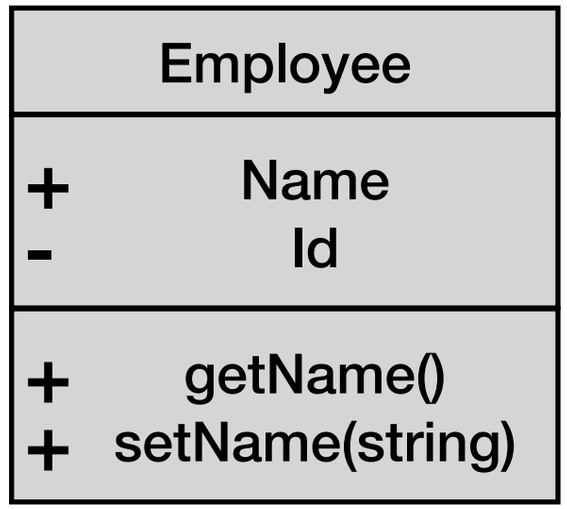
UML Diagram Basics

We represent member access with the following:

- means *private*

+ means *public*

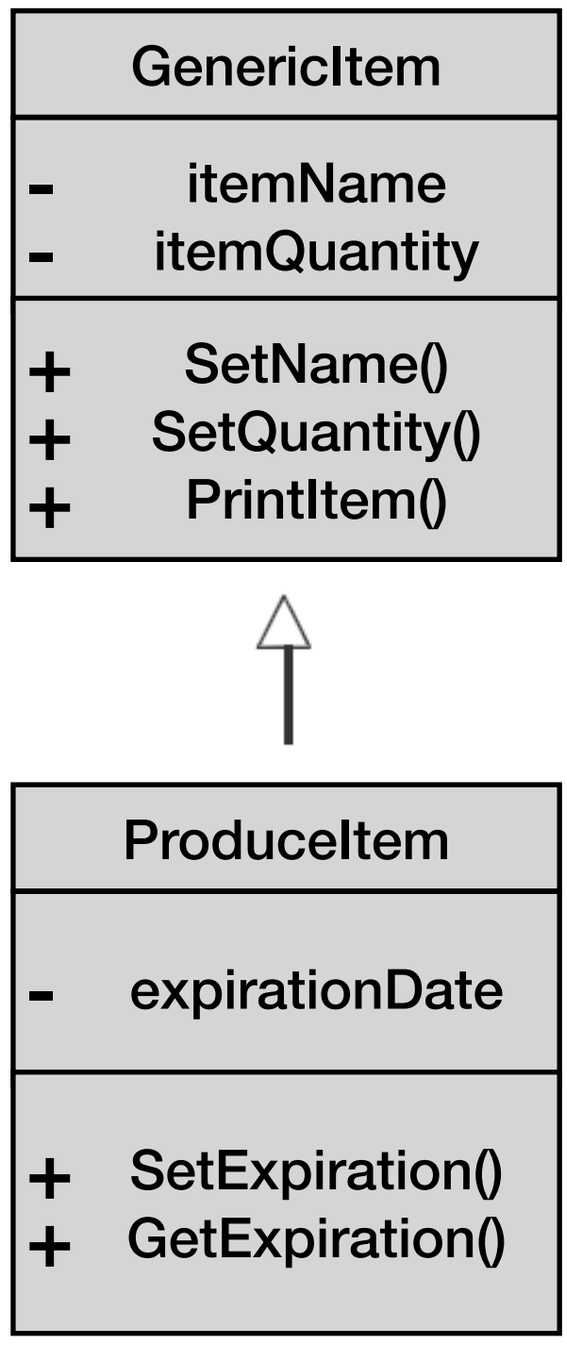
means *protected*



UML Diagram Basics

Defining Relationships

Solid line with unfilled arrow indicates class derivation



Base Class

Derived Class

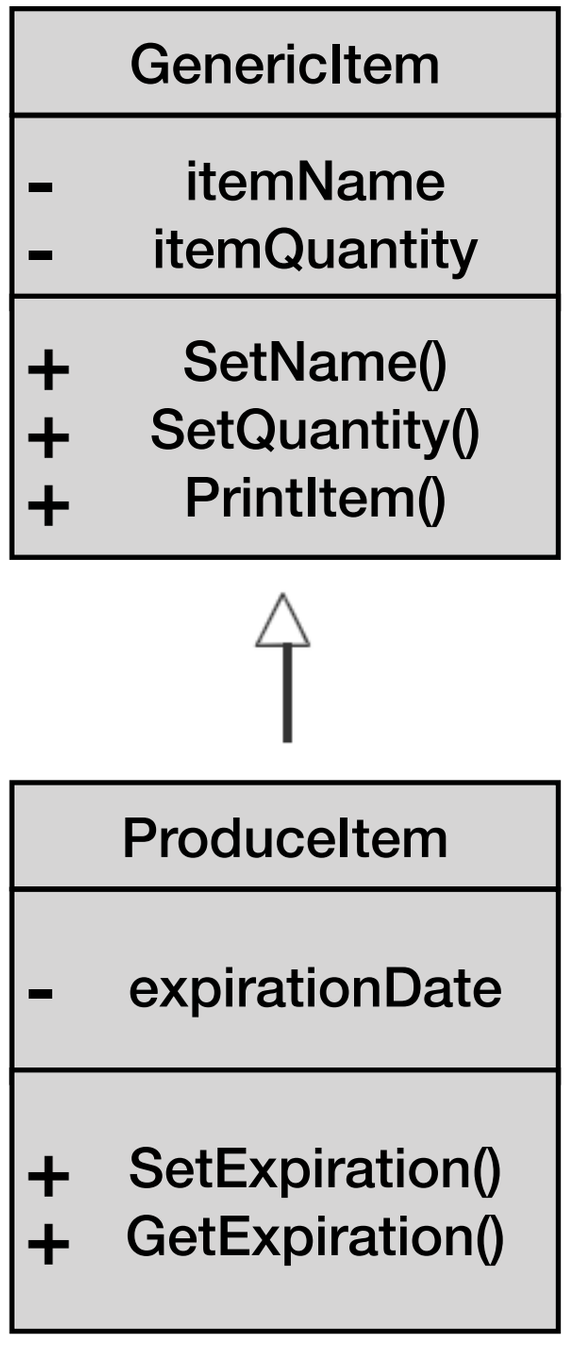
example: produce.cpp

UML Diagram Basics

Defining Relationships

Derived class diagrams only contain data members *not* defined in the Base Class

Note that Derived Class still contains Base Class Data Members and Member Functions



Base Class

Derived Class

https://learn.zybooks.com/zybook/SMUCS1342Spring2020/chapter/13/section/1

Access Specifiers

Three types of **access specifiers:**

Public: class members and functions accessible to any function

Private: class members and functions only accessible by member functions and friends of a class (we will talk about friend functions later)

Protected: class members and functions can be used by member functions or friends of a class, as well as classes derived from the class

Member Access in Base Class

We can specify how a Base Class' members are inherited in the derived class:

	Type of Inheritance		
Base Class Members	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accesible

example: memberAccess.cpp

Life Cycle of Inheritance

Unlike other member functions within a class, **Constructors** and **Destructors** are NOT inherited by a derived class.

Example Constructor:

```
class Employee {
    Employee(): name{"No Name"}, age{0} {
        // body of constructor
    }
};
```

When an object is created:

1. Memory for class is reserved
2. The constructor is called
3. Initializer list is called (if one exists)
4. Body of constructor is executed
5. Control returns to calling process

Life Cycle of Inheritance

Inheritance Constructor:

```
class Person {
    Person() {}
};

class Employee: public Person {
    Employee( ): name{"No Name"}, age{0} {
        // body of constructor
    }
};
```

When an object is created (with inheritance):

1. Memory for class is reserved (*both Derived AND Base class)*
2. The *Derived* constructor is called
3. **The base class is constructed first using appropriate constructor**
4. *Derived* Initializer list is called (if one exists)
5. Body of constructor is executed
6. Control returns to calling process

Life Cycle of Inheritance

```
class Person {
    Person( );
    ~Person( );
};

class Employee: public Person {
    Employee( );
    ~Employee( );
};

Employee employee1; // Create object
```

Lifetime of an object:

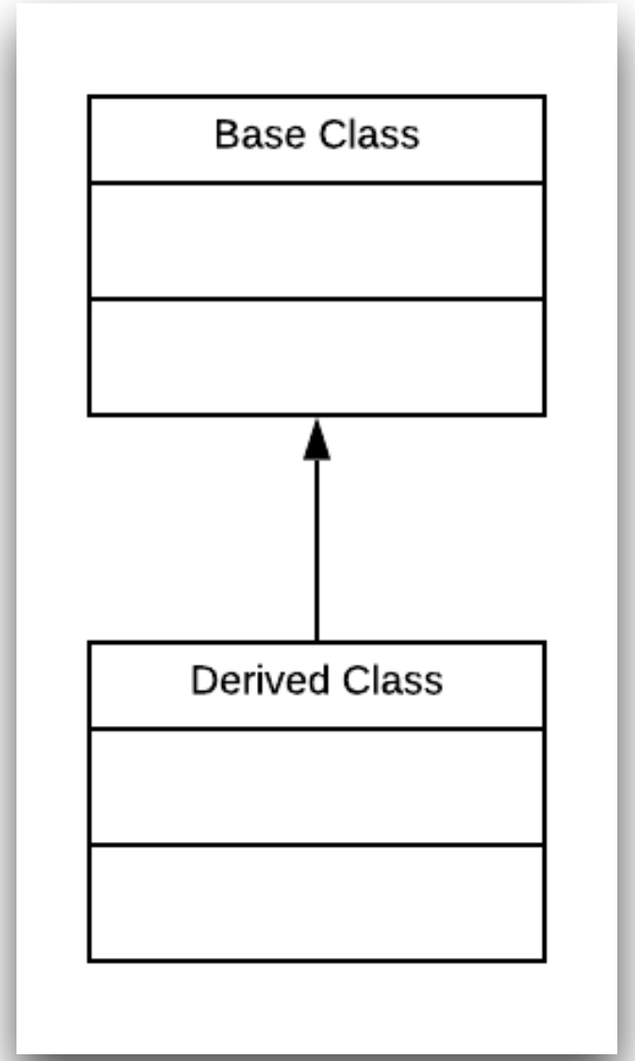
1. Base class constructor
2. Derived class constructor
3. Derived class destructor
4. Base class destructor

Single Inheritance

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

```
class DerivedClass: public Base {

}
```

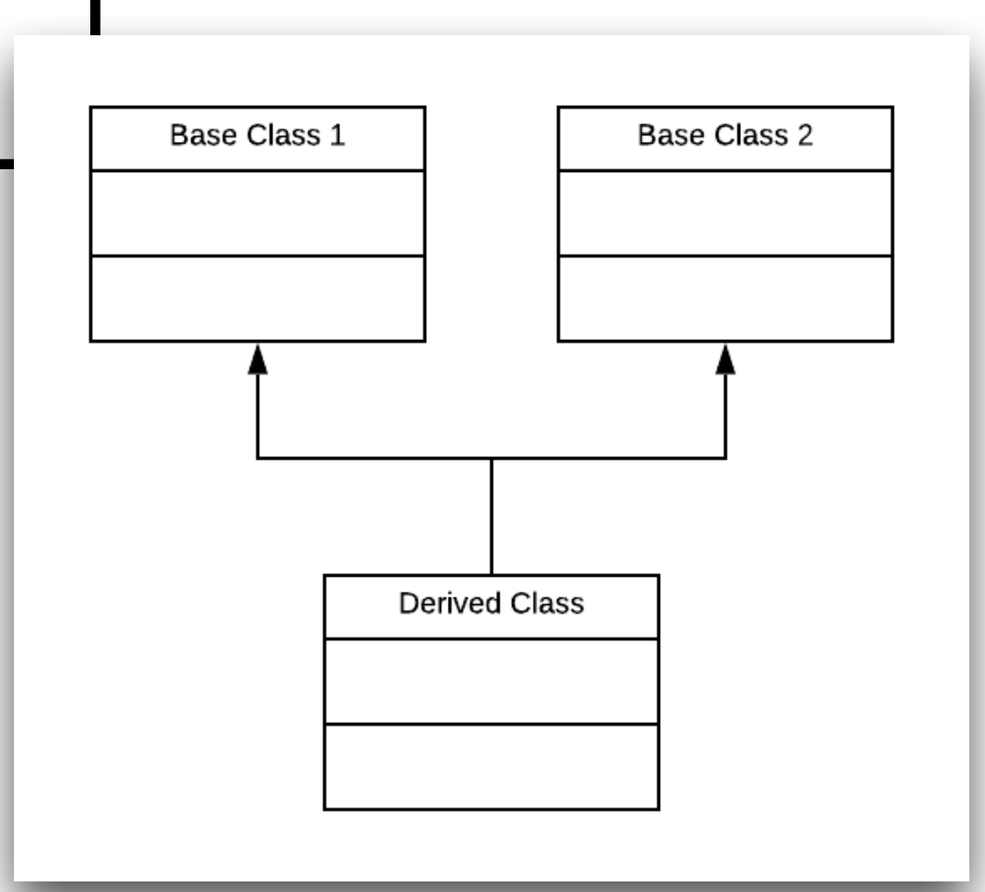


Multiple Inheritance

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.

```
class DerivedClass: public Base1,
    public Base2 {

}
```



Multi-Level Inheritance

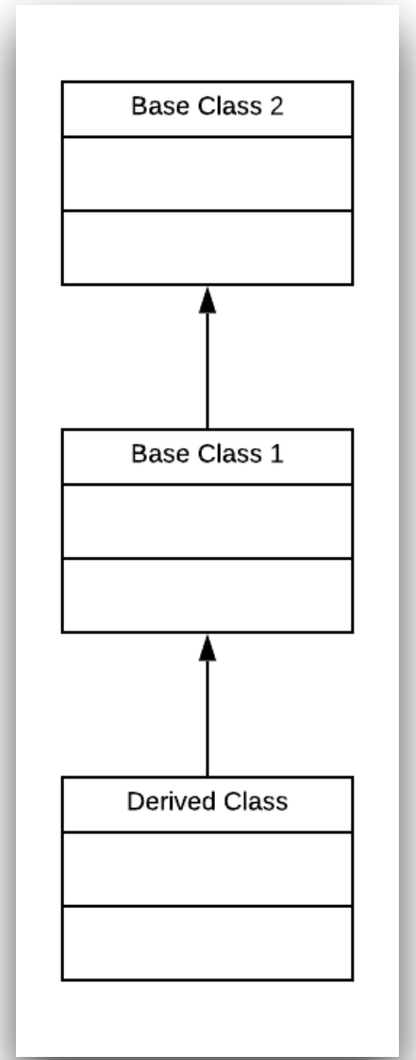
Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

```
class Base1: public Base2 {

}

class DerivedClass: public Base1 {

}
```



Hierarchical Inheritance

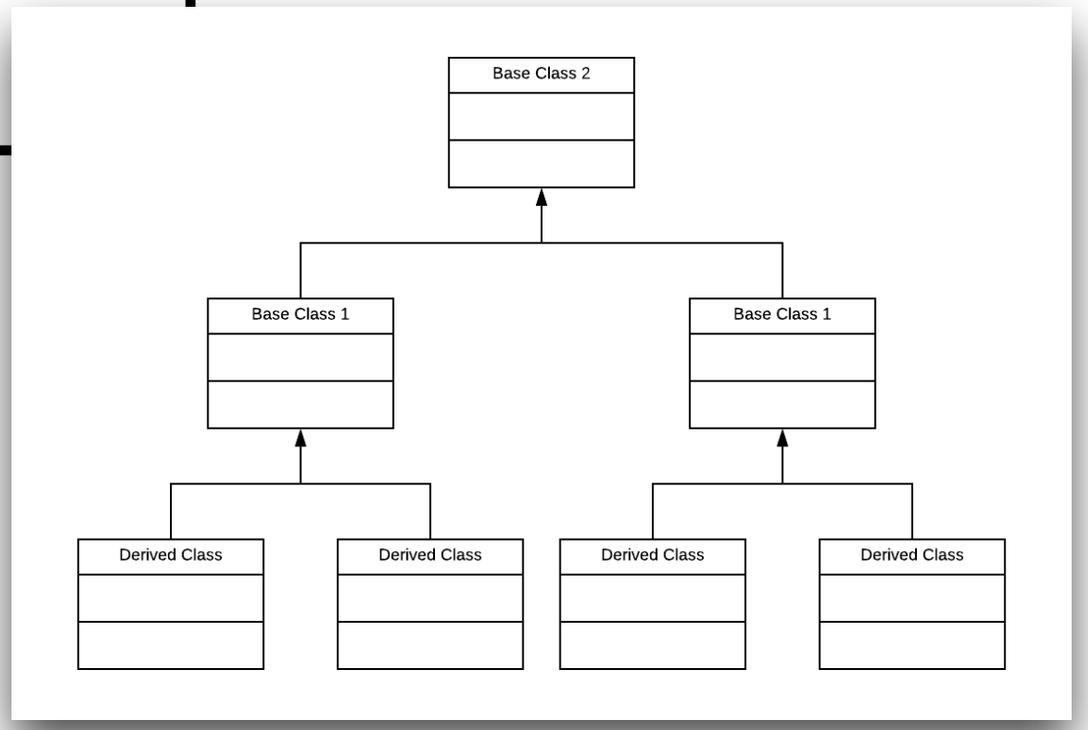
Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

```
class DerivedClass1: public Base {

}

class DerivedClass2: public Base {

}
```



Is-a vs Has-a Relationship

Is-a relationship - an is-a relationship is representative of Inheritance, where an object inherits specific properties from a base class

Has-a relationship - a has-a relationship is when we use Composition to represent the fact that an object is made up of other objects (NOT inheritance).

example: abstract.cpp

Has-A Relationship

In a has-a relationship, there is no inheritance involved. Instead on object is **composed** of another. This relationship is known as Object **Composition.**

```
class ChildInfo {
    string firstName;
    string birthDate;
    string schoolName;

    ...
};

class MotherInfo {
    string firstName;
    string birthDate;
    string spouseName;
    vector<ChildInfo> childrenData;

    ...
};
```

Is-A Relationship

In an is-a relationship, we are using **Inheritance.** Determining whether a set of objects are defined using a has-a relationship vs. an is-a relationship is a helpful strategy for determining if **inheritance** or **composition** is needed.

```
class PersonInfo {
    string firstName;
    string birthDate;

    ...
};
```

```
class ChildInfo : public PersonInfo {
    string schoolName;

    ...
};

class MotherInfo : public PersonInfo {
    string spouseName;
    vector<ChildInfo> childrenData;

    ...
};
```

Friend Functions

A **friend** function of a class is a function defined outside of the scope of the class but has access to all private and protected data members and functions of that class.

Use the *friend* keyword to denote a friend function.

```
class Employee {
   string name;

   public:
      friend void print(Employee employee);
      void setName(string name) {
         this->name = name;
      }
};
```

example: friend.cpp

Friend Class

Similarly to a friend function, a **friend class** is a class who has access to private and protected data members and member functions of another class

Friendships are never corresponded unless specified. Defining one class friendship does not mean the friend class is also a friendly class.

Friendships are *not transitive*, meaning the friend of a friend is not a friend unless explicitly defined.

example: friendClass.cpp

Overriding member functions

Overriding member functions is possible when a derived class defines a member function that has the same name as the base class.

```
class BaseClass {
  void print();
};

class DerivedClass: public BaseClass {
  void print();
};
```

Overriding vs Overloading

Overloading: functions of the same name but with *different* parameter types.

Overriding: functions with the same *signature* in the derived class as the base class will “hide” the base class function.

- We can refer to the base class definition of a function if we provide the base class scope.

```
class DerivedClass: public BaseClass {
  void print() {
    BaseClass::print();
  }
};
```

example: override.cpp

Polymorphism

Polymorphism - refers to the behavior of a function of an object will differ dependent upon the *type* of the object.

Overloading and Overriding are examples of **compile-time polymorphism.**

```
void insert(double value);
void insert(int value);
void insert(char value);
```

Polymorphism

Runtime Polymorphism is when the compiler cannot make the determination of which function to run, and the determination is instead performed while the program is running.

```
class Base {
  virtual void print();
}

class Derived: public Base {
  void print();
}
```

example: polymorphism.cpp

The *virtual* keyword

A **virtual function** is a member function that may be overridden in a derived class and for which runtime polymorphism is used.

Denoted by the keyword: *virtual*

```
class Base {
  virtual void print();
}

class Derived: public Base {
  void print();
}
```

example: virtual.cpp

Pure Virtual Functions

A base class can also have a **pure virtual function.**

This tells the programmer that the derived class *must* override or implement the virtual function for polymorphism.

When a class has a *pure virtual function* this is known as an **abstract class.**

```
class Base {
  virtual void print() = 0;
  virtual int getItem() = 0;
}
```

An abstract class cannot be instantiated but is only used for inheritance and/or polymorphism.

example: abstract.cpp