

## Inheritance

### Access Modifier

1. Public
2. Private
3. Protected

<pre>Public Class A {     Int i , j;      void abc ()     {         S=u+t;     }     Int x , y  }</pre>	<ol style="list-style-type: none"><li>1. Particular member access modifier</li><li>2. Access modifier block</li><li>3. Class declaration as a public</li></ol>
---	--

---

### Concept Of Inheritance

- Way-1 : calling parent class component from main method using child class object reference
- Way-2 : calling parent class member from child class

	Mode of inheritance		
parent class access modifier	Public	Protected	private
Public	w1 ,w 2 Publicly	<del>W1</del> , w2 Protectedly	<del>W1</del> , w2 Privately (2 <sup>nd</sup> generation is locked in multi level )
Protected	<del>W1</del> , w2 protectedly	<del>W1</del> , w2 Protectedly	<del>W1</del> , w2 Privately (2 <sup>nd</sup> generation is locked in multi level )
Private	<del>W1</del> , <del>W2</del> Not possible	<del>W1</del> , <del>W2</del> Not possible	<del>W1</del> , <del>W2</del> Not possible

---

## Mode Of inheritance

- Public Mode
- Private Mode
- Protected Mode

Public mode	Protected mode	Private Mode
<pre>class classOne{ public: };  class classTwo : public classOne{ } };</pre>	<pre>class classOne{ public: };  class classTwo : Protected classOne{ } };</pre>	<pre>class classOne{ public: };  class classTwo : private classOne{ } };</pre>

---

## Types Of Inheritance

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. Hybrid Inheritance
5. Hierarchical Inheritance

1. Single

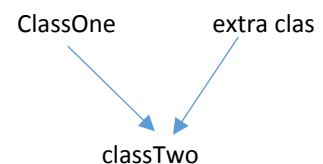
```
class classOne{
public:
};

class classTwo : public classOne{
}
};
```

2. Multiple Inheritance

```
class classOne{
public:
};

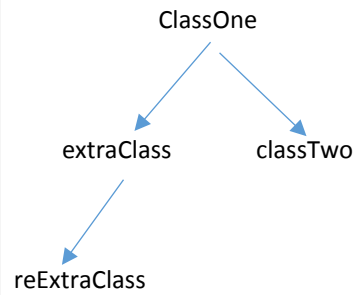
class extraClass{
public:
```



```
};  
  
class classTwo : public classOne , extraClass{  
    }  
};
```

### 3. Multi Level Inheritance

```
class classOne{  
public:  
  
};  
  
class extraClass : public classOne{  
public:  
};  
  
Class reExtra : public extraClass {  
Public:  
};  
  
class classTwo : public classOne {  
    }  
};
```



## Friend class

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes.

We can declare a friend class in C++ by using the **friend** keyword.

class1            class2    class3            .....    classN

Public

Protected

Private

Real life case

- Custom content
- Page - owner/ admin - moderation
- Reliable second mail , id

[NB: details of friend class, friend function – Book ]

## Access Modifiers

1. Public
2. Private
3. Protected

Ways of mentioning or using an access modifier

<pre>Class A {     public :         Int i , j;          void bc ()         {             S=u+t;         }      public:         Int x , y }</pre>	<pre>public Class B {     Int i , j;     void ab ()     {         S=u+t;     }     Int x , y }</pre>	<pre>Class C {     Public Int i ,     Private int j;      Public int ac ()     {         B on;         on.abc();         S=u+t;     }      Int x , y }</pre>	<pre>Int main () {     C obj;     Obj.ac; }</pre>
--	--	--	---

Let us now look at each one of these access modifiers in detail:

**1. Public:** All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

	<pre> public Class B {     Int i , j;     void ab ()     {         S=u+t;     }     Int x , y } </pre>	<pre> Class C {     Public Int i ,     Private int j;      Public int ac ()     {         B on;         on.ab();     }      Int x , y } </pre>	<pre> Int main () {     C obj;     Obj.ac; } </pre>
--	--	--	---

**2. Private:** The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

<pre> Class A { private:     Int i , j;      void bc ()     {         S=u+t;     }     Void nm()     {         Cout &lt;&lt; I         // private         member are </pre>		<pre> Class C {     Public Int i ,     Private int j;      Public int ac ()     {         A on;         on.bc();         // A class member         is private , so not         accessible     } } </pre>	<pre> Int main () {     C obj;     Obj.ac; } </pre>
---	--	--	---

accessible inside the class }		Int x , y  }	
private: Int x , y }			

## Encapsulation (Getter Setter Method)



Encapsulate

Private:

Int pet;

Int speed;

Int getPet()

{}

Int setSet()

{}

# Polimorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

## Types of Polymorphism

### 1. Compile Time Polimorphism

- Method Overloading
- Operator Overloading

### 2. Runtime Polimorphism

- Method overriding
- Virtual function

Based on the functionality, you can categorize polymorphism into two types:

#### ● Compile-Time Polymorphism:

When the relationship between the definition of different functions and their function calls, is determined during the compile-time, it is known as compile-time polymorphism. This type of polymorphism is also known as static or early binding polymorphism. All the methods of compile-time polymorphism get called or invoked during the compile time.

You can implement compile-time polymorphism using function overloading and operator overloading. Method/function overloading is an implementation of compile-time polymorphism where the same name can be assigned to more than one method or function, having different arguments or signatures and different return types. Compile-time polymorphism has a much faster execution rate since all the methods that need to be executed are called during compile time. However, it is less preferred for handling complex problems since all the methods and details come to light only during the compile time. Hence, debugging becomes tougher.

The implementation of compile-time polymorphism is achieved in two ways:

- Function overloading



- Operator overloading

## Runtime Polymorphism

In runtime polymorphism, the compiler resolves the object at run time and then it decides which function call should be associated with that object. It is also known as dynamic or late binding polymorphism. This type of polymorphism is executed through [virtual functions](#) and function overriding. All the methods of runtime polymorphism get invoked during the run time.

Method overriding is an application of run time polymorphism where two or more functions with the same name, arguments, and return type accompany different classes of the same structure. This method has a comparatively slower execution rate than compile-time polymorphism since all the methods that need to be executed are called during run time. Runtime polymorphism is known to be better for dealing with complex problems since all the methods and details turn up during the runtime itself.

The implementation of run time polymorphism can be achieved in two ways:

- Function overriding
- Virtual functions

### **Function Overriding : (reference - Programiz)**

In [C++ inheritance](#), we can have the same function in the base class as well as its derived classes.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

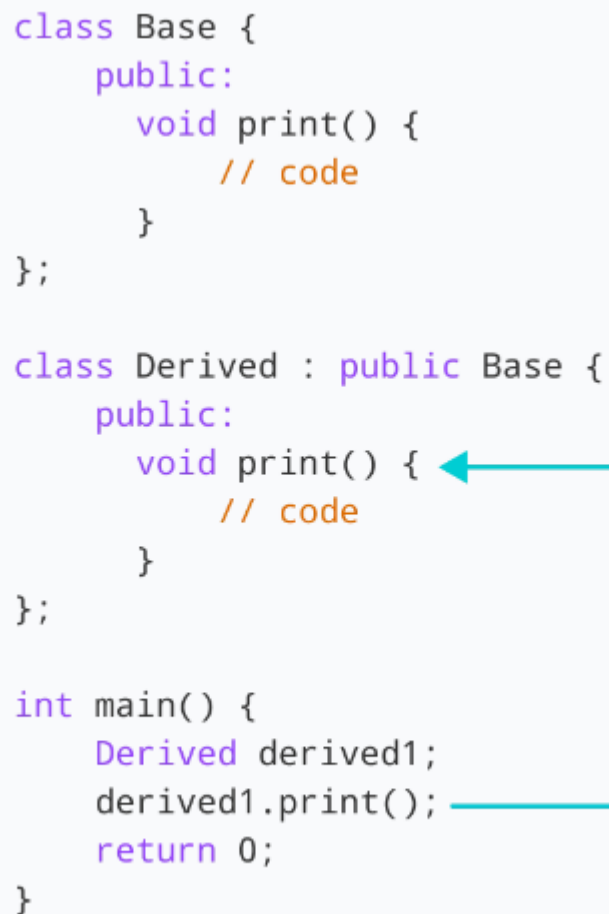
So, different functions are executed depending on the object calling the function.

This is known as **function overriding** in C++. For example,

Example :

1. Animal code (in Program file)
- 2, Code-2

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



Here, the same function `print()` is defined in both Base and Derived classes. So, when we call `print()` from the Derived object `derived1`, the `print()` from Derived is executed by overriding the function in Base.

As we can see, the function was overridden because we called the function from an object of the Derived class. Had we called the `print()` function from an object of the Base class, the function would not have been overridden.

3, Code -3

C++ Access Overridden Function to the Base Class

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1, derived2;  
  
    derived1.print();  
  
    derived2.Base::print();  
  
    return 0;  
}
```

The diagram consists of two teal arrows. The first arrow originates from the `derived1.print();` line in the `main()` function and points to the `void print() {` line inside the `Derived` class. The second arrow originates from the `derived2.Base::print();` line in the `main()` function and points to the `void print() {` line inside the `Base` class.

#### 4, Code- 4

Call overridden function from derived class .

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
            Base::print();  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

The diagram illustrates the execution flow of the provided C++ code. It features three main components: the `Base` class, the `Derived` class, and the `main` function. A red arrow originates from the `derived1.print();` line in the `main` function, pointing to the `void print()` method of the `Derived` class. Another red arrow starts from the `Base::print();` line within the `Derived::print()` method, pointing to the `void print()` method of the `Base` class. This visualizes the call stack where the `main` function calls the `Derived` method, which in turn calls the `Base` method.

5, Code- 5

Calling overridden function using pointer

```
using namespace std;

class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* ptr = &derived1;

    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

## Virtual Function

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

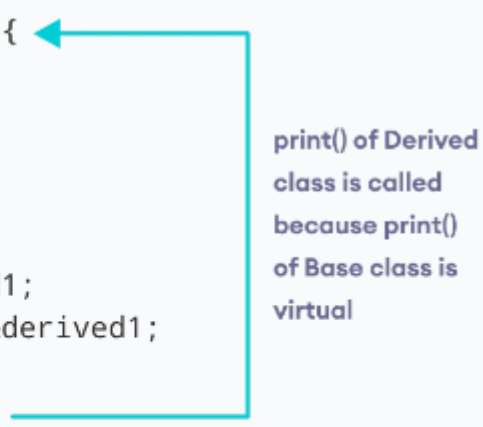
```
class Base {
public:
    virtual void print() {
        // code
    }
};

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

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```



print() of Derived class is called because print() of Base class is virtual

Later, if we create a pointer of Base type to point to an object of Derived class and call the print() function, it calls the print() function of the Base class.

In other words, the member function of Base is not overridden.

In order to avoid this, we declare the print() function of the Base class as virtual by using the virtual keyword.

## C++ override Identifier

C++ 11 has given us a new identifier `override` that is very useful to avoid bugs while using virtual functions.

This identifier specifies the member functions of the derived classes that override the member function of the base class.

For example,

```
class Base {  
    public:  
        virtual void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() override {  
            // code  
        }  
};
```

If we use a function prototype in `Derived` class and define that function outside of the class, then we use the following code:

```
class Derived : public Base {  
    public:  
        // function prototype  
        void print() override;  
};
```

```
// function definition
void Derived::print() {

    // code

}
```

## Use of C++ override

When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.

Using the override identifier prompts the compiler to display error messages when these mistakes are made.

Otherwise, the program will simply compile but the virtual function will not be overridden.

Some of these possible mistakes are:

- **Functions with incorrect names:** For example, if the virtual function in the base class is named `print()`, but we accidentally name the overriding function in the derived class as `pint()`.
- **Functions with different return types:** If the virtual function is, say, of `void` type but the function in the derived class is of `int` type.
- **Functions with different parameters:** If the parameters of the virtual function and the functions in the derived classes don't match.
- No virtual function is declared in the base class.

## Use of C++ Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat`.

Suppose each class has a data member named `type`. Suppose these variables are initialized through their respective constructors.

```
class Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Animal") {}

}
```



```

... ..
};

class Dog : public Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Dog") {}

        ... ..

};

```

```

class Cat : public Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Cat") {}

        ... ..

};

```

Now, let us suppose that our program requires us to create two public functions for each class:

1. `getType()` to return the value of `type`
2. `print()` to print the value of `type`

We could create both these functions in each class separately and override them, which will be long and tedious.

Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```

class Animal {

    ... ..

    public:

```

```

... ..

virtual string getType {...}

};

... ..

```

```

... ..

void print(Animal* ani) {

    cout << "Animal: " << ani->getType() << endl;

}

```

This will make the code **shorter**, **cleaner**, and **less repetitive**.

## Example 2: C++ virtual Function Demonstration

```

// C++ program to demonstrate the use of virtual function

#include <iostream>#include <string>using namespace std;
class Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Animal() : type("Animal") {}

        // declare virtual function
        virtual string getType() {
            return type;
        }
};

class Dog : public Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Dog() : type("Dog") {}

        string getType() override {
            return type;
        }
};

```

```

class Cat : public Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Cat() : type("Cat") {}
        string getType() override {
            return type;
        }
};

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);

    return 0;
}

```

## Output

```
Animal: Animal
```

```
Animal: Dog
```

```
Animal: Cat
```

Here, we have used the virtual function `getType()` and an `Animal` pointer `ani` in order to avoid repeating the `print()` function in every class.

```

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

```

In `main()`, we have created 3 `Animal` pointers to dynamically create objects of `Animal`, `Dog` and `Cat` classes.

```
// dynamically create objects using Animal pointers  
  
Animal* animal1 = new Animal();  
  
Animal* dog1 = new Dog();  
  
Animal* cat1 = new Cat();
```

We then call the `print()` function using these pointers:

1. When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()`.
2. When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()`.
3. When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()`.

## Class Diagram

### UML Class Notation

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory information**.*



#### Class Name:

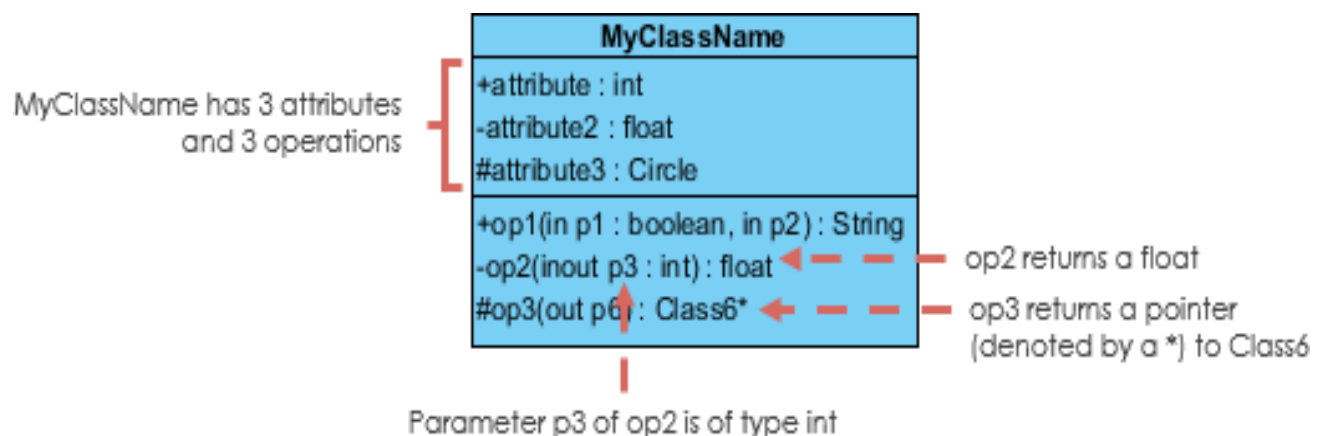
- The name of the class appears in the first partition.

#### Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

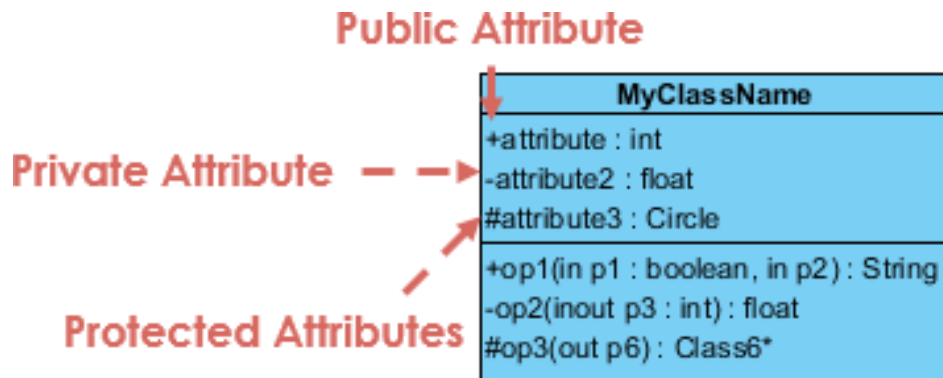
#### Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



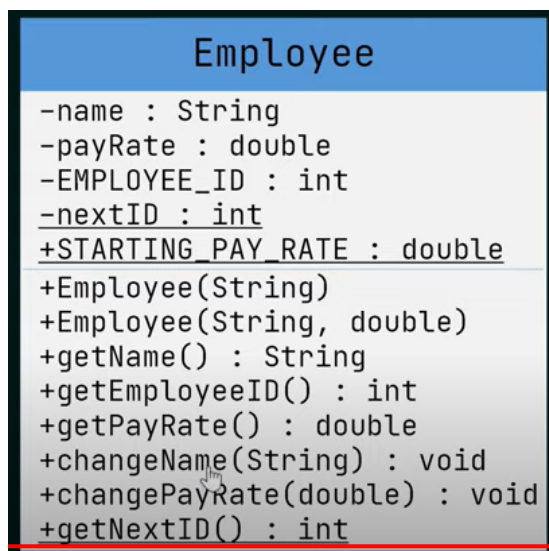
## Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

## Static and Final member



Static - members are underlined (nextID)

Cons - members are written in capital letter (STRATING\_PAY\_RATE )

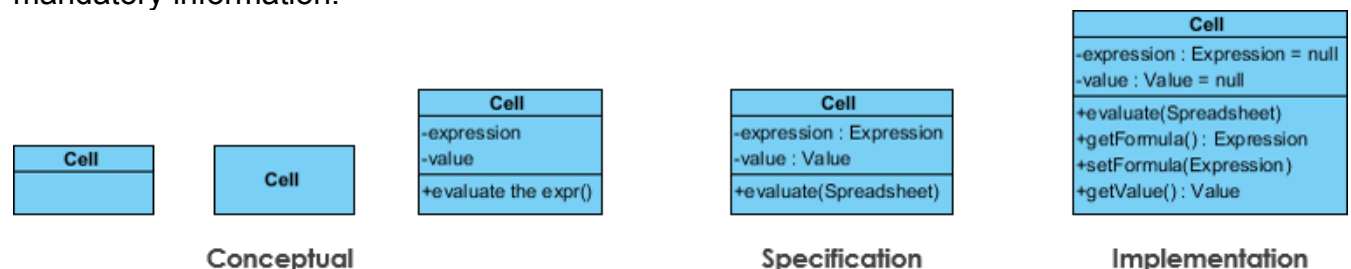
## Perspectives of Class Diagram

The choice of perspective depends on how far along you are in the development process. During the formulation of a **domain model**, for example, you would seldom move past the **conceptual perspective**. **Analysis models** will typically feature a mix of **conceptual and specification perspectives**. **Design model** development will typically start with heavy emphasis on the **specification perspective**, and evolve into the **implementation perspective**.

A diagram can be interpreted from various perspectives:

- **Conceptual**: represents the concepts in the domain
- **Specification**: focus is on the interfaces of Abstract Data Type (ADTs) in the software
- **Implementation**: describes how classes will implement their interfaces
- 

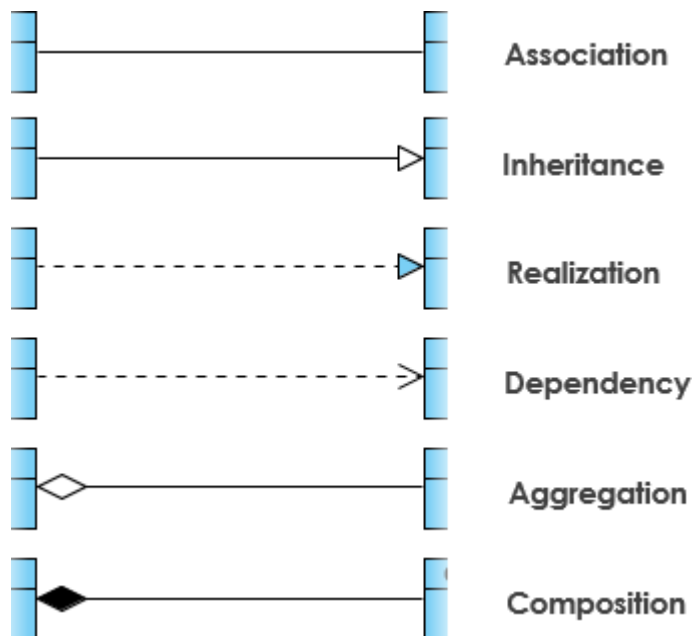
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



## Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

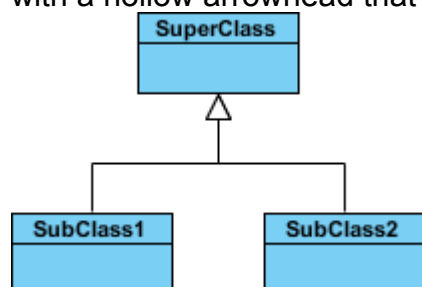


## Inheritance (or Generalization):

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

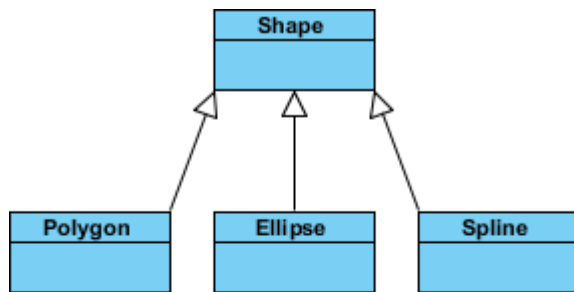
The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.



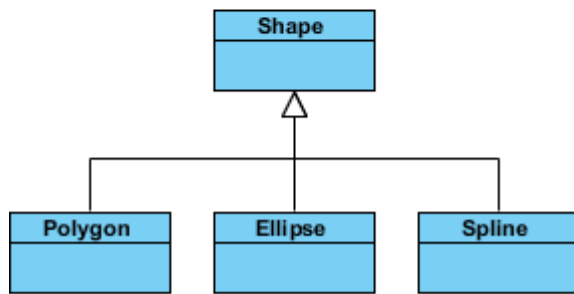
## Inheritance Example - Shapes

The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.





**Style 1: Separate target**



**Style 2: Shared target**

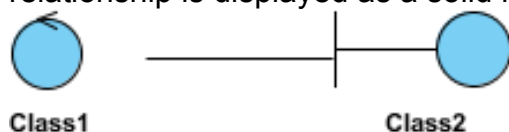
## Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

### Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2

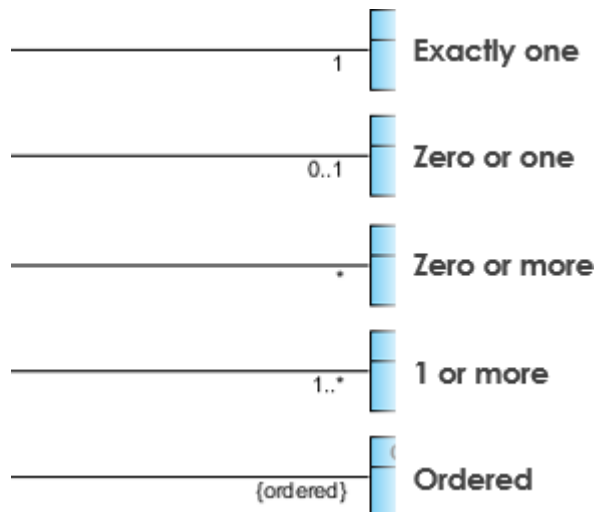
The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



## Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

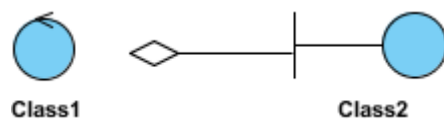


## Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the \*) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



## Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.



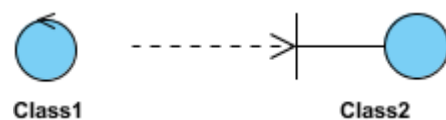
## Dependency

An object of one class might use an object of another class in the code of a method.

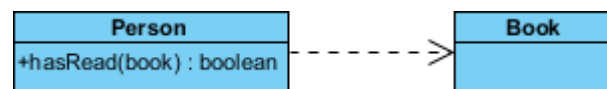
If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



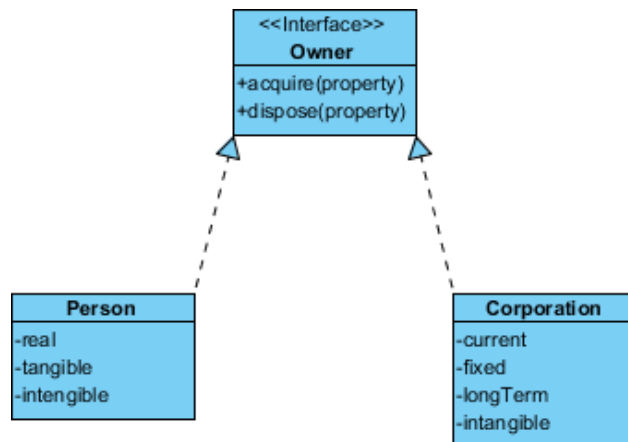
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



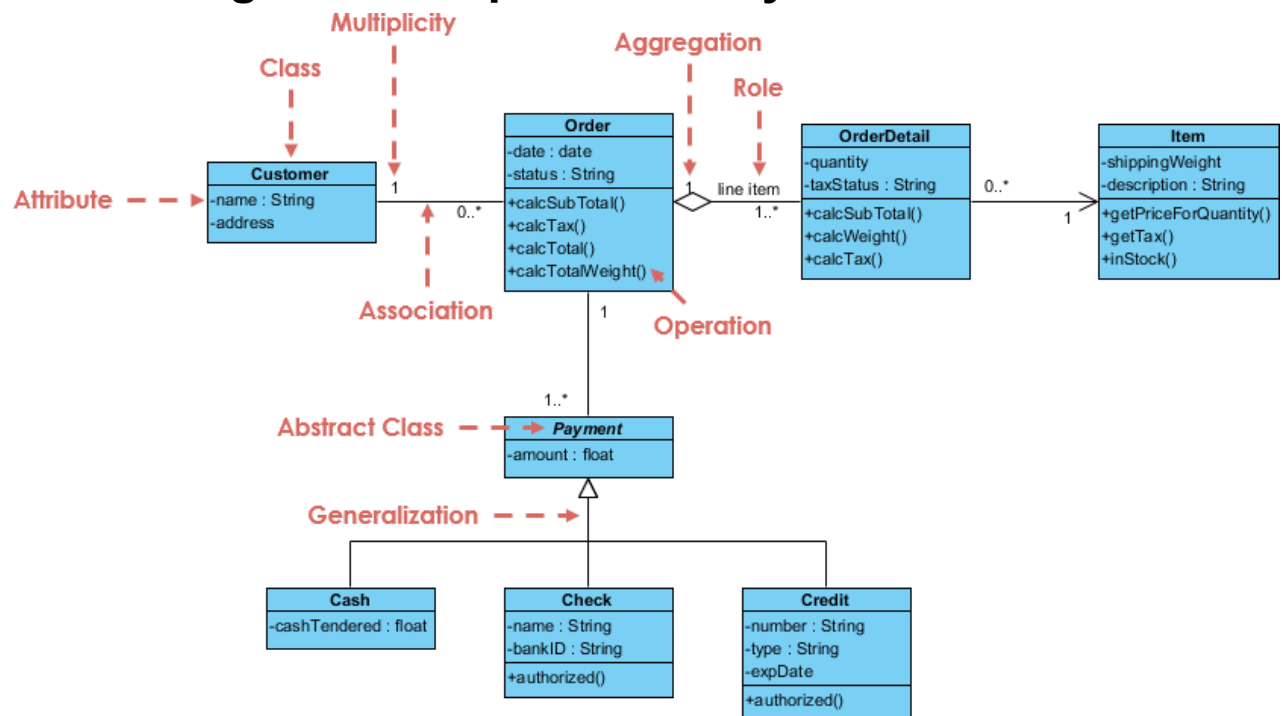
## Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.

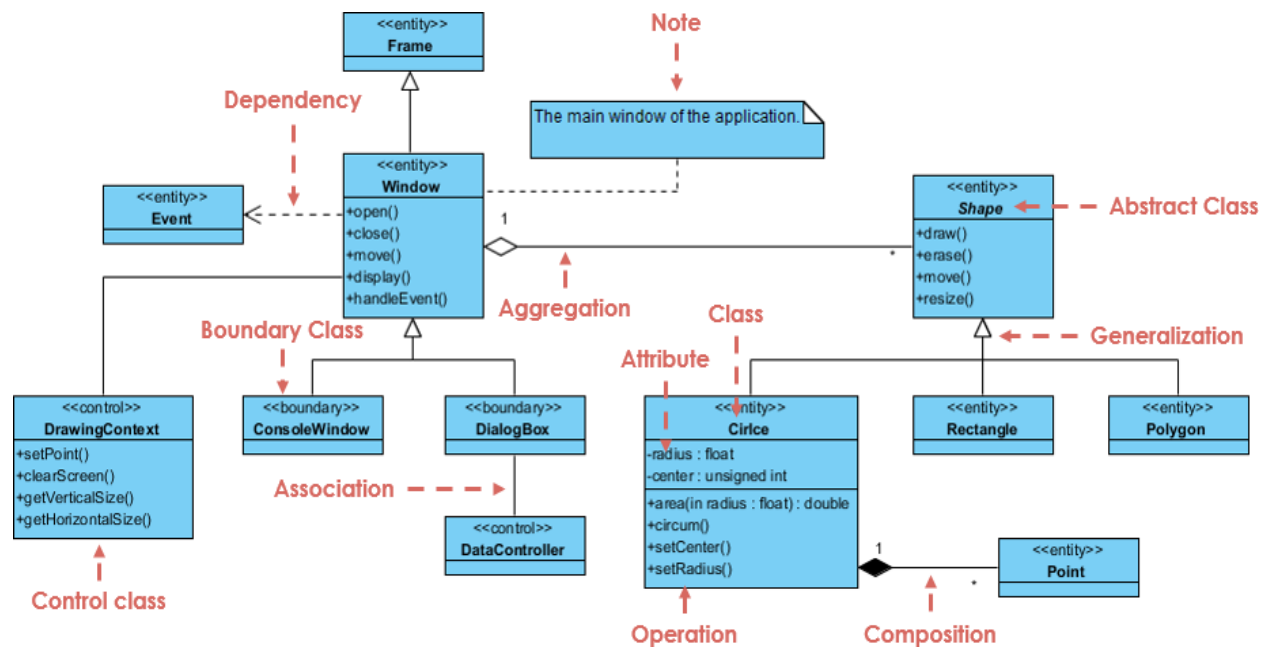


## Class Diagram Example: Order System



## Class Diagram Example: GUI

A class diagram may also have notes attached to classes or relationships.



## Resources : (Class Diagram)

1. For study -

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

2. For Practice -

[OOP Exercises - Java Programming Tutorial \(ntu.edu.sg\)](http://www.ntu.edu.sg)

(Posted in google classroom)

3. Some Question :

- Draw a class diagram for a library management system

-----

## File System / File Handle

1. Connect with files
  2. Open a file from your code
  3. Open an existing from code
  4. Writing on a file
  5. Read from an existing file documents
  6. Changing file document
  7. Search in a file
  8. Alphabetically search
  9. Count
- 

Modes of opening a file :

1. Read - r
2. Write - w
3. Append - a