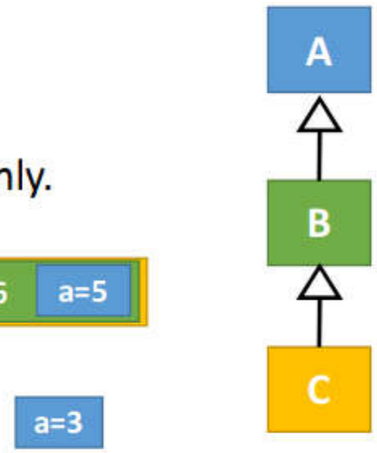
Polymorphism

Object-Oriented Programming

Compile Time Binding or Static Binding

- We can assign derived class object to base class object.
- Only copy data of base class, derived portion is discarded.
- Compile time binding system will call base class functions only.

```
void main(){
    A a1 (2);
    B b1 (3, 4);
    C c1 (5,6,7);

a1 = b1; //slice b1 and copy in a1 the A's portion only.
a1.print(); //Base print called print A's data only.
a1.funb(); a1.print(3); //Compile time Error

a1 = c1; //slice c1 and copy in a1 the A's portion only.
a1.print(); //Base print called print A's data only.
a1.funb(); a1.func();   //Compile time Error
a1.print(3); a1.print(3, 4); //Compile time Error

}
```

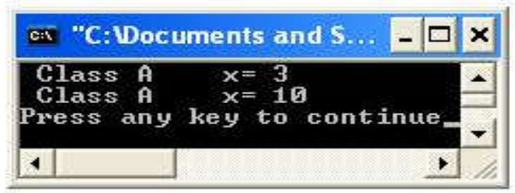


a=3

a=5

One of the key features of **class inheritance** is that a pointer to a derived class is **type-compatible** with a pointer to its base class. ***Polymorphism*** is the art of taking advantage of this simple but powerful and versatile feature.

```
class A
{
public:
    int x;
    void print() { cout<<" Class A    x= "<<x<<endl;}
};

class B: public A
{
public:
    int a;
    void print() {  cout<<" Class B    a= "<<a; A::print();}
};

void printAll(A &obj)
{
    obj.print();
}
void main()
{
    A obj1;
    obj1.x=3;
    B obj2;
    obj2.x=10;
    obj2.a=100;

    printAll(obj1);
    printAll(obj2);
}
```




```
class A
{
public:
    int x;
    virtual void print() { cout<<" Class A    x= "<<x<<endl;}
};

class B: public A
{
public:
    int a;
    void print() {  cout<<" Class B    a= "<<a; A::print();}
};

void printAll(A &obj)
{
    obj.print();
}
void main()
{
    A obj1;
    obj1.x=3;
    B obj2;
    obj2.x=10;
    obj2.a=100;

    printAll(obj1);
    printAll(obj2);
}
```

The word “**polymorphism**” means that the one and same class may show many (“poly”) forms (“*morphs*”) not defined by the class itself, but by its subclasses.

Another definition says that polymorphism is **the ability to realize class behavior in multiple ways**.

To understand the true nature of the phenomena, I believe that an example will work better.

```cpp
class Pet{
protected:
    string name;
public:
    Pet(string n){
        name=n;
    }

    virtual void MakeSound(){
        cout << name <<" the Pet says nothing";
    }
};
```

```cpp
class Cat: public Pet{
    public:
        Cat(string n):Pet(n) {   }

        void MakeSound(){
            cout<<name<<" the Cat says: Meow! Meow!"<<endl;
        }
};
```

```cpp
class Dog: public Pet{
    public:
        Dog(string n):Pet(n){  }

        void MakeSound(){
            cout<<name<<" the Dog says: Woof! Woof!"<<endl;
        }
};
```

```cpp
int main()
{
    Pet *pet1, *pet2;
    Cat *myCat;
    Dog *myDog;

    myCat=new Cat("Kitty");
    pet1=myCat;
    pet1->MakeSound();
    myCat->MakeSound();


    myDog=new Dog("Doggie");
    pet2=myDog;
    pet2->MakeSound();
    myDog->MakeSound();

    cout<<endl<<endl;
    return 0;
}
```

Output:

```
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
```

```cpp
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};
```

```cpp
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Output:
20
10

So what we need?
We need some kind of MAGIC, through which area() of **Rectangle** or **Triangle** can be called having just pointers of the base class.

Member **area** could have been accessed with the *pointers* to **Polygon** if area were a member of Polygon instead of a member of its derived classes. But the problem is that **Rectangle** and **Triangle** implement different **versions of area**, therefore there is not a single common version that could be implemented in the base class.

Ref: http://www.cplusplus.com/doc/tutorial/polymorphism/

```
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b; }
11     virtual int area ()
12       { return 0; }
13 };
14
15 class Rectangle: public Polygon {
16   public:
17     int area ()
18       { return width * height; }
19 };
20
21 class Triangle: public Polygon {
22   public:
23     int area ()
24       { return (width * height / 2); }
25 };
```

A class that declares or inherits a virtual function is called a *polymorphic class*.

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Output:
20
10
0

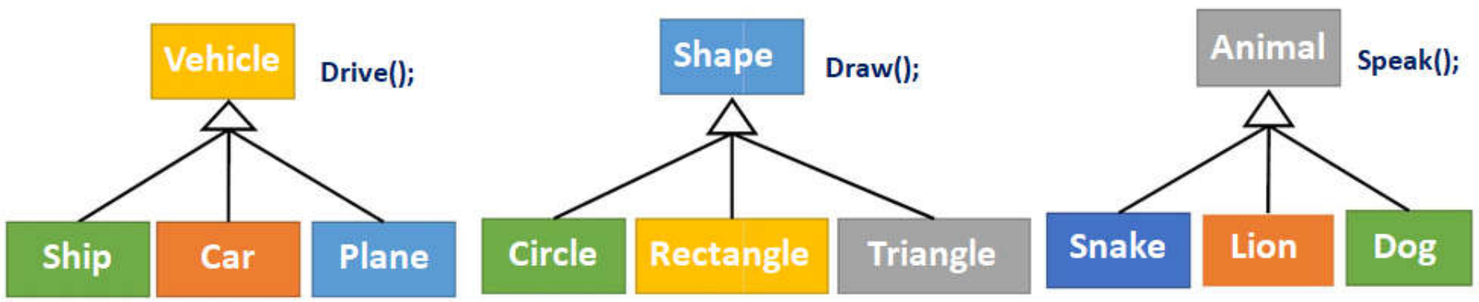
What will happen if we remove virtual keyword in Polygon class?

Do you remember slicing problem?

Polymorphism

Polymorphism is a feature of **OOP** that allows the object to behave differently in different conditions.

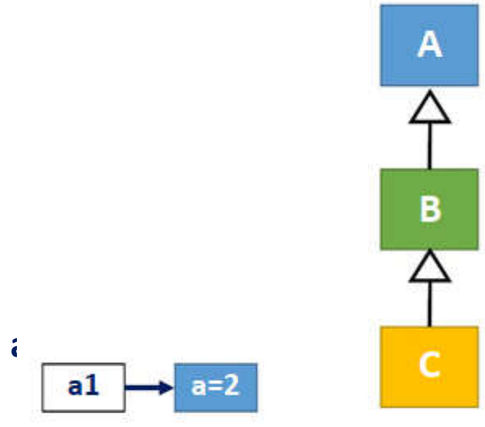
- **Poly** means “Many” **Morphism** mean “Forms/Shapes”
  - Same base type behavior will be changed according to object of derived.
  - Use objects without knowing their types explicitly.
  - Extend the program with more functionalities through derive classes.
  - Need one single array of base class to collect all different objects of derived.
  - Base class represent a larger set for all objects (base and derived)

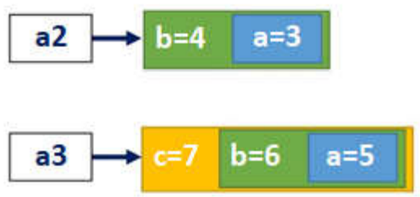


Polymorphism

- One object can show different behaviors.

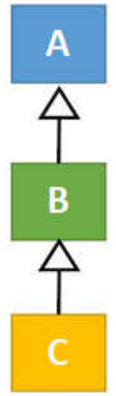
```
void main(){
    A * a1 = new A(2); //A's pointer to A's object
    a1->print(); //A's print called.

    A * a2 = new B(3, 4); //A's pointer to B's object
    a2->print(); //Should call B's print

    A * a3 = new C(5, 6, 7); //A's pointer to C's object
    a3->print(); //Should call C's print
}
```





Polymorphism

1. Only base class inherited functions can be called through base pointer.
2. Override base class function in derived classes.
3. Change Compile time binding of functions to Run time binding,
    1. **Run time binding:** Call functions according to object type not pointer type.
    2. Make functions virtual in base class.
        - Inherited as virtual in all derived classes, no need to make virtual again.
        - All virtual functions binding change to runtime.

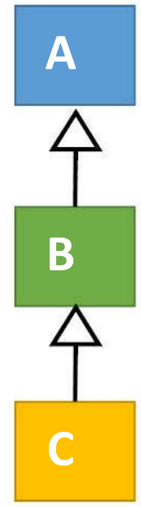



```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
};
```

Runtime Binding

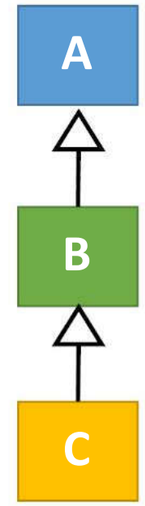
```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
};
```

```
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
//override print function inherited from A
    virtual void print(){
        A::print();
        cout<<b;
    }
//overload print function inherited from A
    void print(int x){ cout<<x+b; }
    void funb() { cout<<“funb”<<endl};
};
```



Runtime Binding…

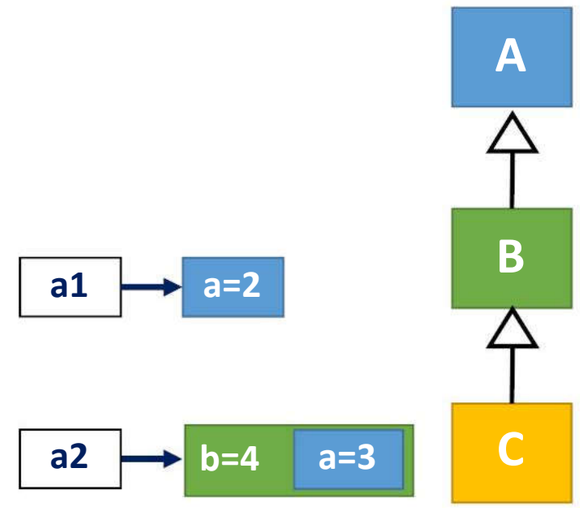
```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
//override print function inherited from B
 virtual void print() {
        B::print();
        cout<<c;
    }
//overload print function inherited from B
    void print(int x, int y){
        cout<<x+y+c;
    }

    void func(){ cout<< “func” <<endl; }
};
```



Runtime Binding…

- Call the function according to the type of object not pointer.

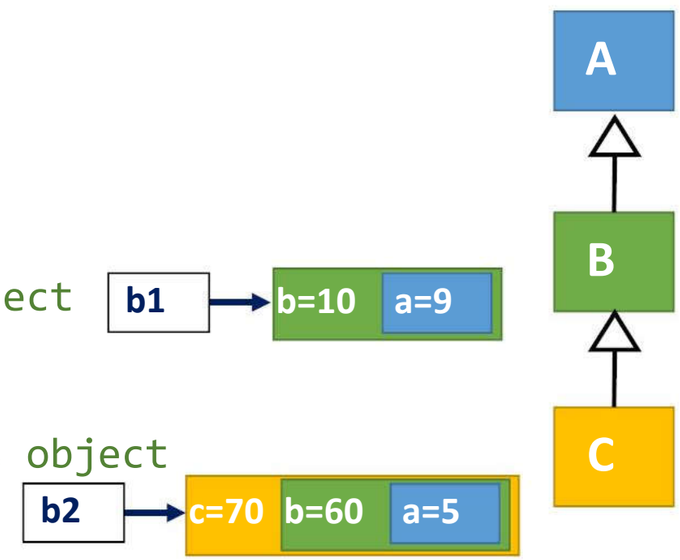
```
void main(){
    A * a1 = new A(2); //A's pointer to A's object
    a1->print(); //A's print called.

    A * a2 = new B(3, 4); //A's pointer to B's object
    a2->print(); //B's print called
    a2->funb(); a2->print(3); //Compile time Error

    A * a3 = new C(5, 6, 7); //A's pointer to C's object
    a3->print(); //C's print called
    a3->funb(); a3->print(3); //Compile time Error
    a3->func(); a3->print(3,8); //Compile time Error
}
```



Runtime Binding…

- Call the function according to the type of object not pointer.

```
void main(){
    B * b1 = new B(9, 10); //B's pointer to B's object
    b1->print(); //B's print called.

    B * b2 = new C(5, 60, 70); //B's pointer to C's object
    b2->print(); //C's print called.
    b2->funb(); b2->print(3);

    b2->func(); b2->print(3,8); //Compile time Error

    B * b3 = new A(2); //Error: B's pointer to A's object
    //Every derived is a base but every base is not a derived.
    //Allowed if explicit cast made
}
```



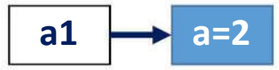
Runtime Binding…

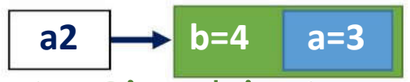
Base class static object		Call base class functions only
Derived class static object		Call derived class functions & inherited functions.
Base class static object	Derived class static object	Call base class functions only. **Slicing Issue** only copies base data in base object
Derived class static object	Base class static object	Error: Explicit cast required
Base class pointer or reference	Base class object	Call base class functions only
Base class pointer or reference	Derived class object	**Call derived class overridden functions that exist in base.**
Derived class pointer or reference	Base class object	Error: Explicit cast required
Derived class pointer or reference	Derived class object	Call derived class functions & inherited functions.

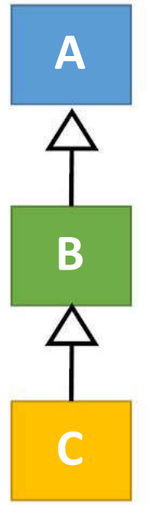
Inheritance (is-a) **Virtual Destructor**

- Destructor should be called according to object type.
- Derived class may have some dynamic members need to deallocate.

```
void main(){
    A * a1 = new A(2); //A's pointer to A's object
    a1->print(); //A's print called.

    A * a2 = new B(3, 4); //A's pointer to B's object
    a2->print(); //B's print called

    A * a3 = new C(5, 6, 7); //A's pointer to C's object
    a3->print(); //C's print called

    delete a1; //A's destructor called
    delete a2; //A's destructor called, should call b's
    delete a3; //A's destructor called, should call c's
}
```



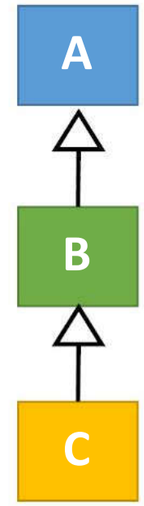




Inheritance (is-a) **Virtual Destructor**

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print(){
        A::print();
        cout<<b;
    }
    virtual ~B(){}

};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
//override print function inherited from B
    void print(){
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```

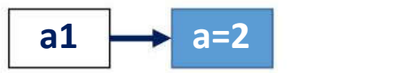


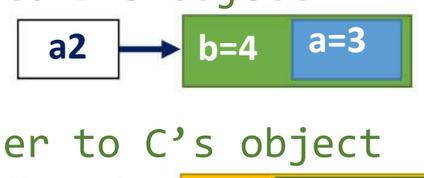
- **Make destructor virtual too.**
- **Destructor is not inherited so make it virtual in all classes.**

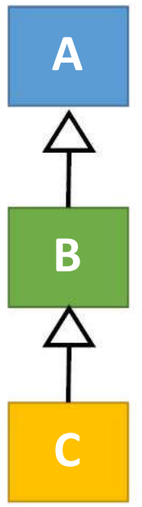
Inheritance (is-a) **Virtual Destructor**

- Destructor should be called according to object type.

```
void main(){
    A * a1 = new A(2); //A's pointer to A's object
    a1->print(); //A's print called.

    A * a2 = new B(3, 4); //A's pointer to B's object
    a2->print(); //B's print called

    A * a3 = new C(5, 6, 7); //A's pointer to C's object
    a3->print(); //C's print called

    delete a1; //A's destructor called
    delete a2; //B's destructor called, which also calls A's
    delete a3; //C's destructor called, which also calls B's then A's
}
```



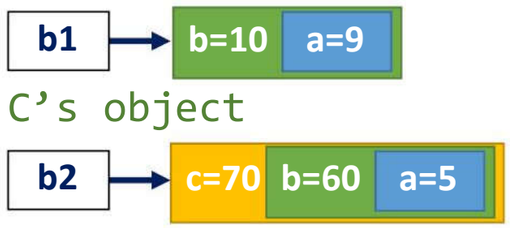


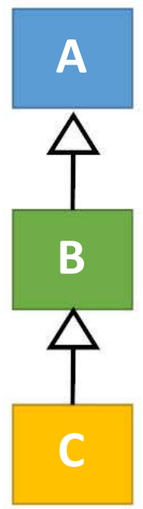


Inheritance (is-a) **Virtual Destructor**

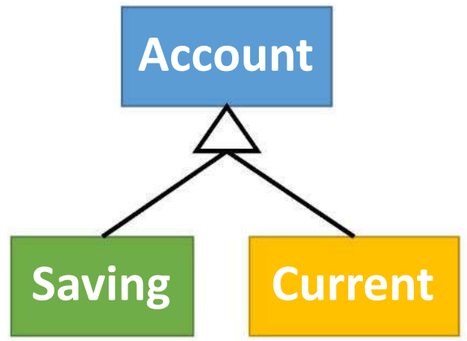
- Call the function according to the type of object not pointer.

```
void main(){
    B * b1 = new B(9, 10); //B's pointer to B's object
    b1->print(); //B's print called.

    B * b2 = new C(5, 60, 70); //B's pointer to C's object
    b1->print(); //C's print called.

    B * b3 = new A(2); //Error: B's pointer to A's object
    //Every derived is a base but every base is not a derived.
    //Allowed if explicit cast made

    delete b1; //B's destructor called, which also calls A's
    delete b2; //C's destructor called, which also calls B's then A's

}
```





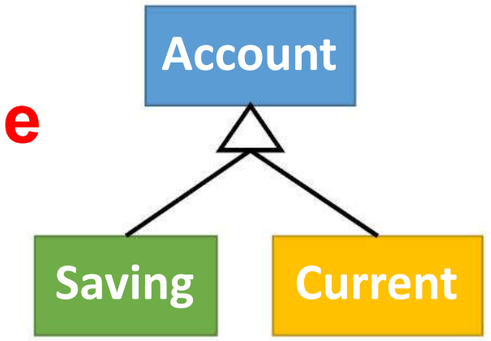
Inheritance (is-a) **Polymorphism Example**




```
class Account{
    int accountNo;
    float amount;
public:
//override Debit and Credit functions according to derived classes.
    virtual void debit(float);
    virtual void credit (float);
    virtual void print(){
        cout<< accountNo ;
        cout<< amount <<endl;
    }
    virtual ~Account(){}
};
```

```
class Current: public Account{
    float serviceCharges;
    float minBalance;
public:
    virtual void print() override{
        Account::print();
        cout<<serviceCharges;
    }
    virtual ~Current(){}
    //Use inherited Debit and Credit
};
class Saving: public Account{
    float interestRate;
public:
    virtual void print() override{
        Account::print();
        cout<<interestRate;
    }
    virtual ~Saving(){}
    //override Debit and Credit
    virtual void debit(float) override;
    virtual void credit (float) override;
};
```

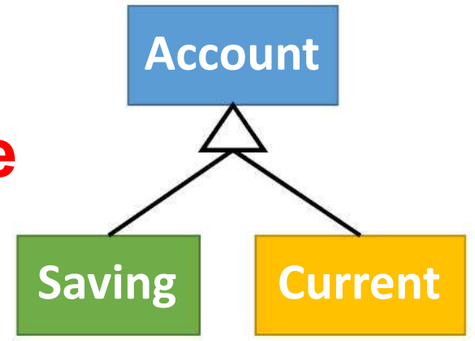
Inheritance (is-a) **Polymorphism Example**



- Call the function according to the type of object not pointer.
- Debit, credit and print function are different for different accounts.

```
void main(){
    Account * a1 = new Saving; //Base pointer to derived object
    a1->print(); //Saving's print called.
    a1->debit(300); //Saving's Debit called.
    a1->credit(900); //Saving's credit called.

    Account * a2 = new Current; //Base pointer to derived object
    a2->print(); //Current's print called.
    a2->debit(500); //Current's Debit called.
    a2->credit(30085); // Current's credit called.

    delete a1; //Saving destructor called, then Account
    delete a2; //Current destructor called, then Account
}
```

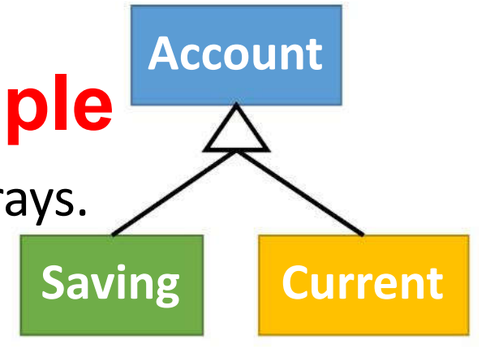
Inheritance (is-a) **Polymorphism Example**



- Maintain a single array of Account instead of two separate arrays.

```
void main(){
//Array of base pointers
    Account ** alist = new Account*[10];
    alist[0] = new Saving;
    alist[1] = new Current;
    alist[2] = new Account;
    ….
//Print data of all accounts polymorphic behavior
    for(int i=0; i<10 ;i++)
        alist[i]->print();

//credit and debit polymorphic behavior
    alist[0]->credit(50);
    alist[2]->debit(333);
}
```

Inheritance (is-a) **Polymorphism Example**



- Maintain a single array of Account instead of two separate arrays.

```
//Destructors show polymorphic behavior
//Destructors are called according to object type
//Destroy all accounts
    for(int i=0; i<10 ;i++)
        delete alist[i];

//Deallocate array of pointers
    delete [] alist;
```

Inheritance (is-a) **Polymorphism Example**

- Create a payroll program for 3 types of employees, paid monthly
  - Salaried (fixed salary, no matter the hours)
  - Hourly (overtime [>40 hours] pays time and a half)
  - Commission (paid percentage of sales)

1. Each employee's pay will be calculated in different way.

2. Override calculatePay function in all employees accordingly.

3. Convert the binding to Runtime in employee class.

```
virtual float calculatePay();
```