Object-Oriented Programming and Exception Handling

- Introduction of Object-Oriented Programming
- Object-Oriented Programming in Java
- Object-Oriented Programming in C++
- Exception handling in Java
- Exception Handling in C++

Object-Oriented Concepts

- Classes
- Objects
- Inheritance
- Encapsulation
- Polymorphism

- Many object-oriented programming (OOP) languages
  - Some only support OOP paradigms (e.g., Java)
  - Some also support procedural programming (e.g., C++)

Object-Oriented Programming Language: Java

- Abstract class
- Interface
- Polymorphism
- Nested class

Abstract Classes

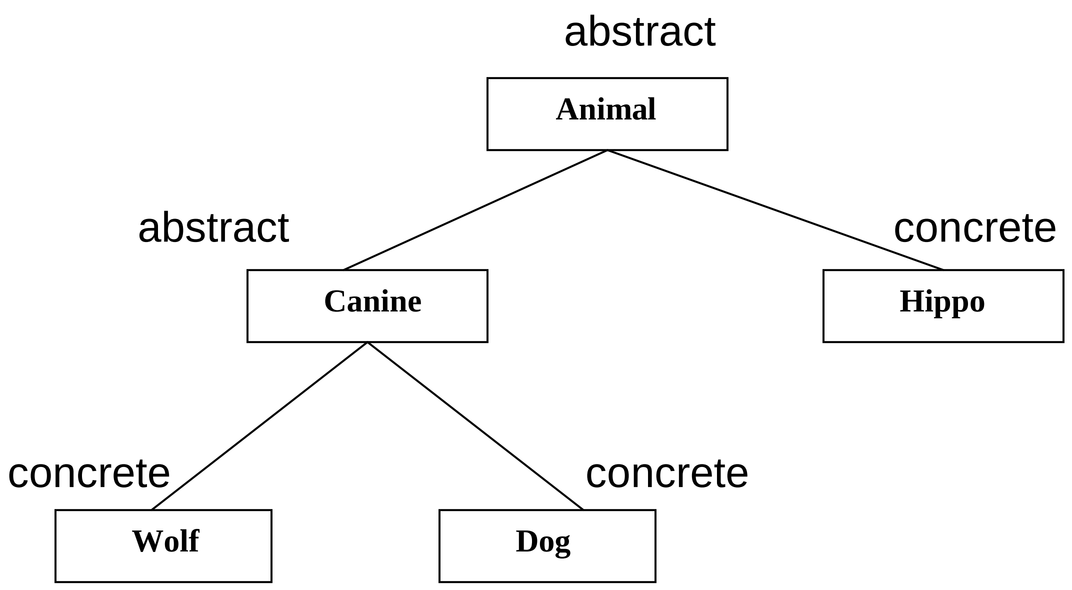
Sometimes, a class that you define represents an abstract concept that should not be instantiated.

The Number class in the java.lang package represents the abstract concept of numbers. It makes sense to model numbers in a program, but it doesn't make sense to create a generic number object.

Instead, the Number class makes sense only as a parent class to classes like Integer and Float, both of which implement specific kinds of numbers.

```
public abstract class Shape {
  abstract int getArea();
}

public class Rectangle extends Shape{
  double ht = 0.0;
  double wd = 0.0;

  public double getArea()
  {

   return (ht*WD);
  }
}
```

```
public class Circle extends Shape{
  double r =0.0;

  public double getArea()
  {

   return (2 * 3.14 * r);
  }

}
```



```
public abstract class Animal  {
  private String name;
  public Animal(String nm)
  { name=nm; }
  public String getName()  // regular method
  { return (name); }
  public abstract void speak(); // abstract
                                  //method - note no {}
}
```

- The only reason to establish this common interface is so it can be expressed differently for each different subclass. It establishes a basic form, so you can say what’s in common with all the derived classes.
- If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don’t (and you may choose not to), then the derived class is also **abstract**.
- Abstract class express only the interface, not a particular implementation, so creating an object of abstract class makes no sense

Interface – more abstract than abstract class

- **A Java *interface* is a collection of abstract methods and constants.**
- An interface permits no implementation, not even partial implementation.
- **An interface is a function specification about what a class do, not how it does.**

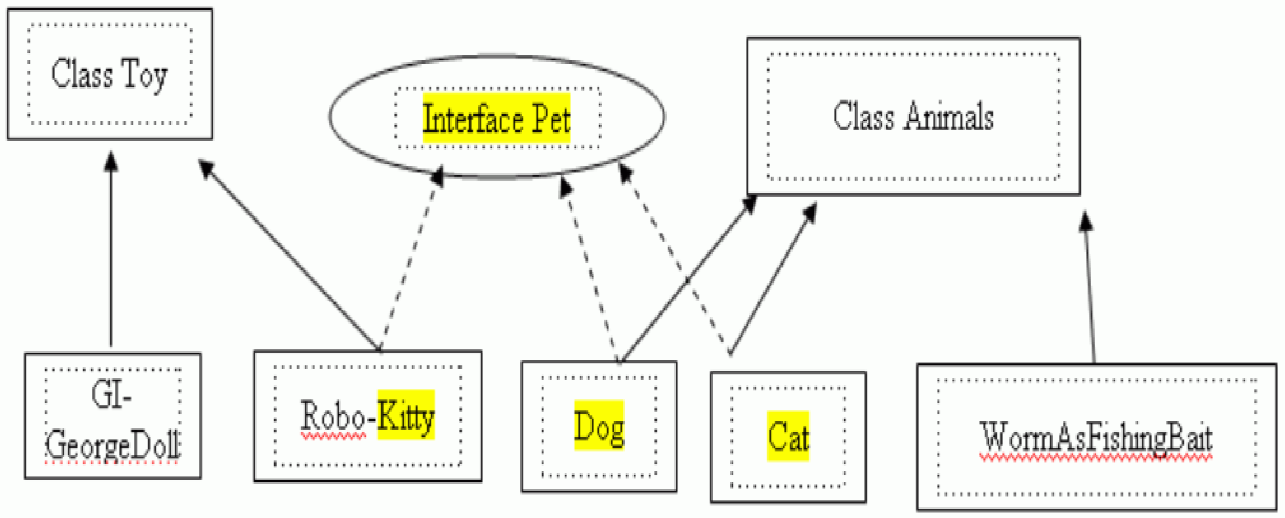
```
public interface Import {
 public double calculateTariff();
}

public class importCar implement Import{
  public double calculateTariff() {
……}}

public class importFood implement Import{
  public double calculateTariff() {
……}}

public class importCloth implement Import{
  public double calculateTariff() {
……}}
```

Abstract class and Interface

An interface cannot implement any methods, whereas an abstract class can.

A class can implement many interfaces but can have only one superclass.

An interface is not part of the class hierarchy. Unrelated classes can implement the same interface. An *interface* defines a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy.



`Robo-Kitty`, `Dog`, and `Cat` implement the `Pet` interface. The `Pet` interface specifies pet-like behavior (methods), such as `lookHappyWhenPetted`.
Robo-Kitty does not implement the Animal behavior of `eatFood`.

Instances of the `WormAsFishingBait` class do not have the `lookHappyWhenPetted` behavior but do have the Animal behavior of `eatFood`.

The `Pet` interface provides a specification for what any class, whether a subclass of `Animal` or a subclass of `Toy`, must do to provide the capability of being a pet.

Implementing Interfaces

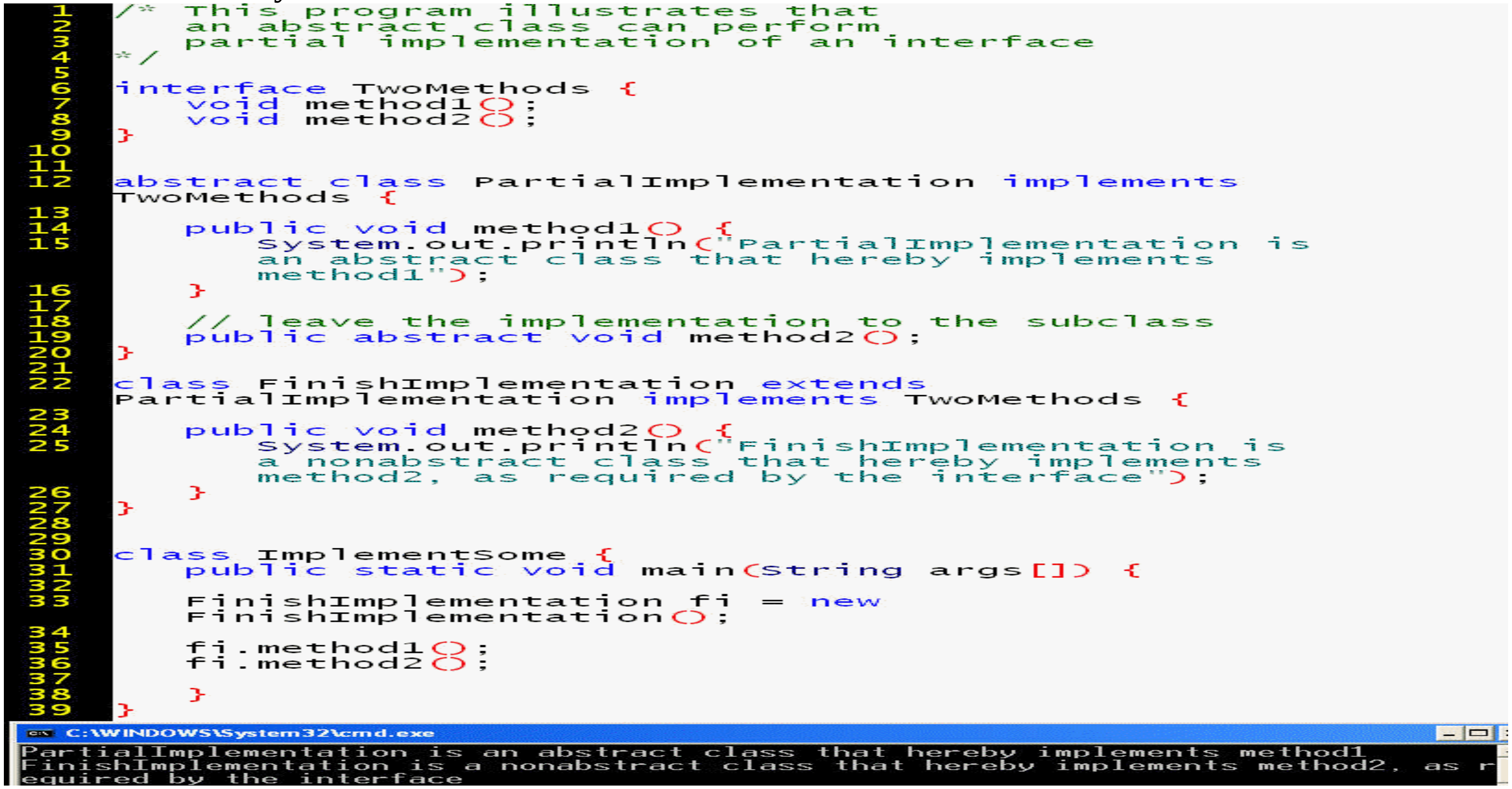
- If a class asserts that it implements an interface, it must define all methods in the interface
- The implementation can be blocks of empty code, but at least in form, each class must have an implementation.

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }
   public void doThis2 (float value,
char ch){ // whatever ……}
   public boolean doTheOther (int
num){  // whatever ……}
}
```

Combining an abstract class with an interface

What if you only want to implement *some* (but not all) of the classes in the interface?
In this case, the implementing class must be declared an abstract class. The duty of implementation must be fulfilled by whatever class extends the abstract class.



```
/*  This program illustrates that
    an abstract class can perform
    partial implementation of an interface
*/

interface TwoMethods {
    void method1();
    void method2();
}

abstract class PartialImplementation implements
TwoMethods {

    public void method1() {
        System.out.println("PartialImplementation is
        an abstract class that hereby implements
        method1");
    }

    // leave the implementation to the subclass
    public abstract void method2();
}

class FinishImplementation extends
PartialImplementation implements TwoMethods {

    public void method2() {
        System.out.println("FinishImplementation is
        a nonabstract class that hereby implements
        method2, as required by the interface");
    }
}

class ImplementSome {
    public static void main(String args[]) {

    FinishImplementation fi = new
    FinishImplementation();

    fi.method1();
    fi.method2();

    }
}
```

C:\WINDOWS\System32\cmd.exe

```
PartialImplementation is an abstract class that hereby implements method1
FinishImplementation is a nonabstract class that hereby implements method2, as r
equired by the interface
```

Extending an interface

You can write an interface that inherits from an existing interface, and the keyword remains **extends**.

```
//One interface can extend another

interface A{
   void meth1();
   void meth2();
}

interface B extends A {
   void meth3();
}

class myClass implements B {
    public void meth1(){
       System.out.println("Implement meth1()");}
    public void meth2(){
       System.out.println("Implement meth2()");}
    public void meth3(){
       System.out.println("Implement meth3()");}
}
```

```
Class myClassTest {
    public static void main (string args[]) {
         myClass ob = new myclass();
          ob.meth1();
          ob.meth2();
          ob.meth3();
    }
}
```

Polymorphism

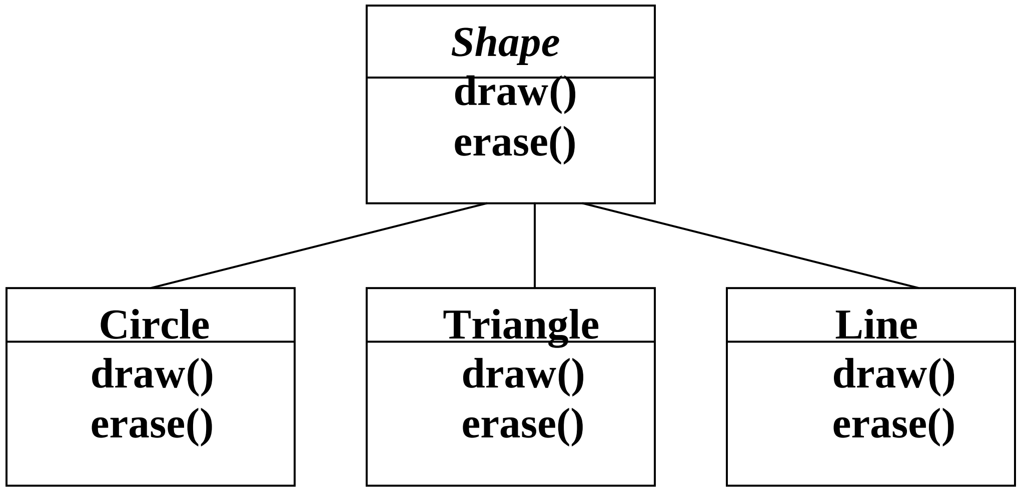
polymorphism means "having many forms", it is based on the late binding technique.

Java defers method binding until run time -- this is called _dynamic binding_ or _late binding_

Java's use late binding that allows you to declare an object as one class at compile-time but executes based on the actual class at runtime.

Polymorphism via Inheritance

```
Shape s = new Circle();
Shape s = new Triangle();
Shape s = new Line();
```

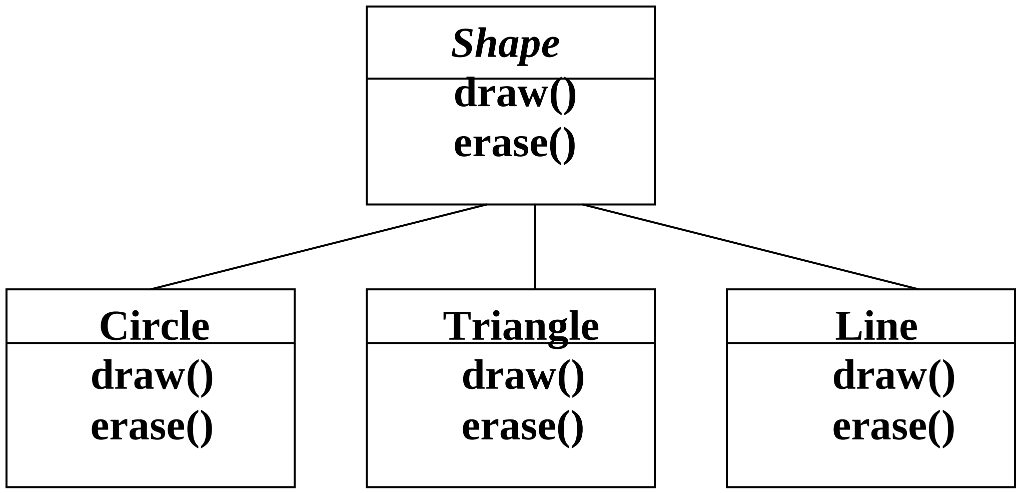


Upcast

- The process of treating a child class as though it were its parent class.
- Considered to be a widening conversion, and can be performed by simple assignment

When you call one of the parent class methods (that have been overridden in the child classes), it is the type of the object being referenced, not the reference type, that determines which method is invoked.

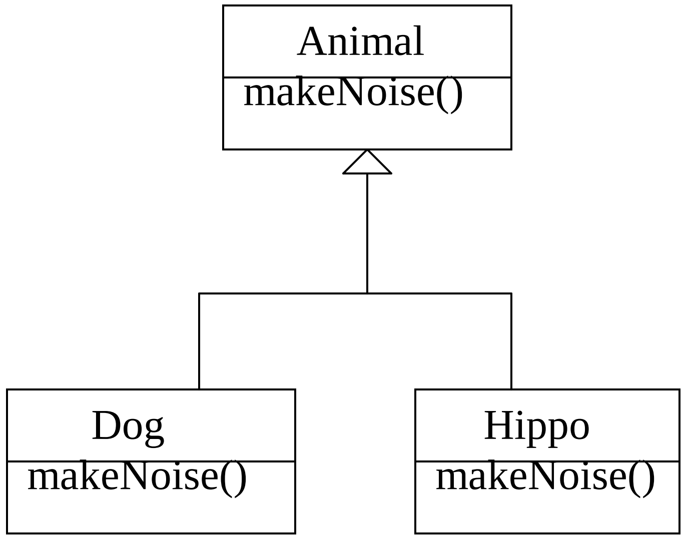
```
Shape s = new Circle();
    s.draw();
```



You can have polymorphic arguments and return type

Animal a = d   and   Animal a = h

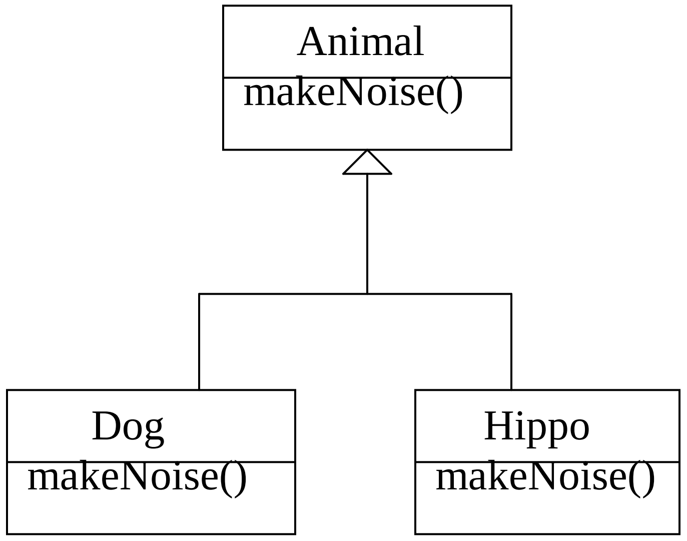
```
class Vet {
      public void giveShot(Animal a) {

            a.makeNoise();
}
}

class PetOwner {
      public void main() {
            Vet v = new Vet() ;
            Dog d = new Dog();
            Hippo h = new Hippo() ;
            v. giveShot (d) ;
            v. giveShot (h);
      }
}
```



Why Polymorphism?

- You can write your code to talk to the parent class and know that all the child classes will work correctly using the same code.
- Easy to maintain your program

```
class Vet {
      public void giveShot(Animal a) {

            a.makeNoise();
}
}

class PetOwner {
      public void main() {
            Vet v = new Vet() ;
            Dog d = new Dog();
            Hippo h = new Hippo() ;
            v. giveShot (d) ;
            v. giveShot (h);
      }}
```



Nested Classes

Non-static nested classes (inner class)

(1) In Java, a non-static nested class is a class nested within another class:

```
class C {
        class D {
        }
}
```

(2) Objects of the non-static nested class are attached to objects of the outer class

```
C c = new C()
D d = c.new D()
```

(3) Because the non-static nested class is considered part of the implementation of the outer class, it has access to *all* of the outer class's instance variables and methods.

```
public class EnclosingClass
{
   private String someMember = "someMember";

   private class InnerClass
   {
      public void doIt()
      {
         System.out.println(someMember);
      }
   }

   public static void main(String[] args)
   {
      new EnclosingClass().new InnerClass().doIt();
   }
}
```

Static Nested classes

If a class is defined within another class and it is marked with the static modifier, it is called a static nested class.

```
public class EnclosingClass{
  private static String staticMember = "Static Member";
  private String someMember = "Some Member";

  static class NestedClass   {
    public void doIt()
    {
      System.out.println(staticMember);
    }
  }

  public static void main(String[] args)
  {
    new NestedClass().doIt();
  }
}
```

(1) They are exactly like classes declared outside any other class

(2) To create object of static nested class, you don't need to create an instance of the enclosing class first.

(3) You cannot access instance members of that enclosing class like Inner Class could previously. A static nested class cannot refer directly to instance variables or methods defined in its enclosing class - it can use them only through an object reference.

(4) A static nested class can access static members (even private ones) of its enclosing class.

Reexportation in C++

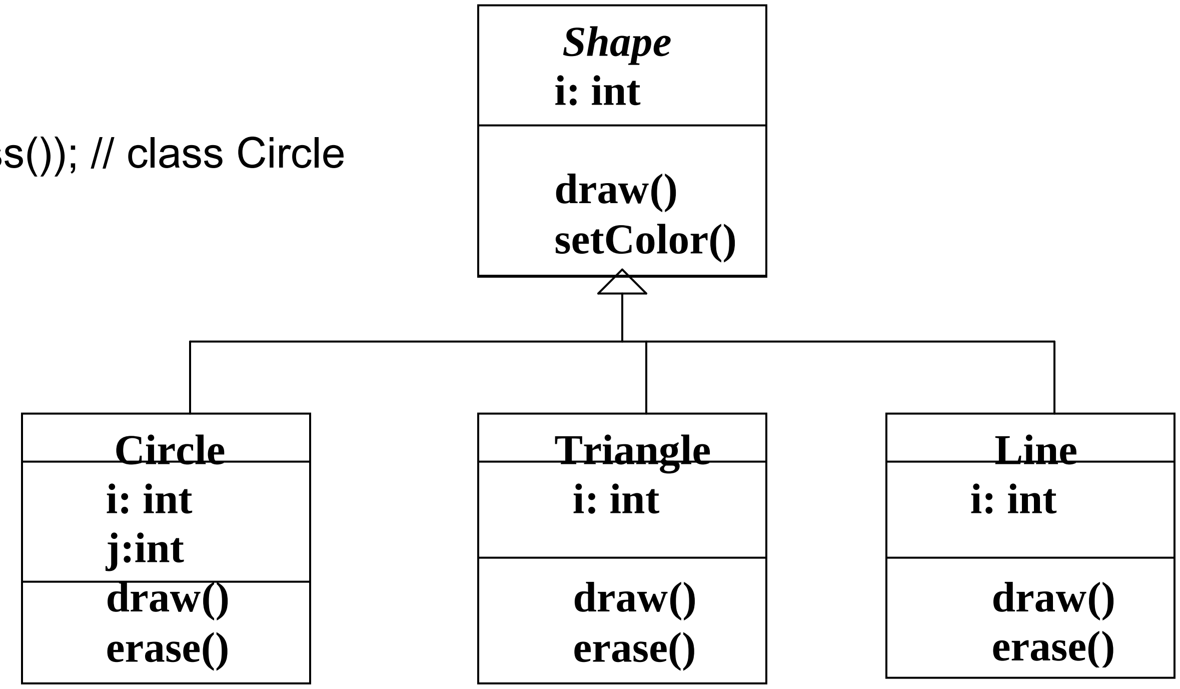
- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (`::`), e.g.,

```
class subclass_3 : private base_class {
        base_class :: c;

...
}
```

Polymorphism with methods/instance variable

- When you call one of the parent class methods draw() (that have been **overridden** in the child classes), the method executed by s.draw() depends on the actual type of the object that s refers to, not on the type of the variable s.
- Overridden variable depends on the type of variable of s

```
Shape s = new Circle();
System.out.println(s.getClass()); // class Circle
s.draw();

int t = s.i;
int k= s.j;//??
s.erase(); //???
s.setColor();
```



Polymorphism in C++

A pointer to a derived class is type-compatible with a pointer to its base class.

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

class CRectangle: public CPolygon {
 public:
   int area ()
     { return (width * height); }
 };
```

```
class CTriangle: public CPolygon {
 public:
   int area ()
     { return (width * height / 2); }  };
int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,  5);
 ppoly2->set_values (4,  5);
 cout << rect.area() << endl; ////???
 cout << trgl.area() << endl;   ///  ???
 return 0;}
```

This is a problem.

- In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes.
- Because CRectangle and CTriangle implement different versions of area, we cannot implement it in the base class. This is when virtual members become handy.

Virtual members

- A member of a class that can be redefined in its derived classes.
- To declare a member of a class as virtual, we must precede its declaration with the keyword virtual.
- Usually has a different functionality in the derived class
- A function call is resolved at run-time

virtual function is a primary tool for polymorphic bevaviour.

The difference between a non-virtual c++ member function and a virtual member function

- the non-virtual member functions are resolved at compile time. This mechanism is called ***static binding***.
- C++ virtual member functions are resolved during run-time. This mechanism is known as ***dynamic binding***.

```
class CPolygon {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b; }
   virtual int area ()
     { return (0); }  };
```

```
20
10
0
```

```
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Abstract base classes

Abstract base classes are very similar to virtual member.

Virtual member: we can defined a valid function.
Abstract base classes: no implementation at all.

This is done by appending =0 (equal to zero) to the function declaration.

```
// abstract class CPolygon
class CPolygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
   { width=a; height=b; }
  virtual int area () =0;};
```

This type of function is called a *pure virtual function*, and all classes that contain at least one pure virtual function are *abstract base classes*.

The main difference between an abstract base class and a regular polymorphic class is that we cannot create objects of it.

We can create pointers to abstract base class and take advantage of all its polymorphic abilities.

```
CPolygon poly;
```

not valid because tries to instantiate an object.

Nevertheless, the following pointers:

```
CPolygon * ppoly1;
CPolygon * ppoly2;
```

would be perfectly valid.

CPolygon is an abstract base class. Pointers to this abstract base class can be used to point to objects of derived classes.

```
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };
```

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

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;}
```

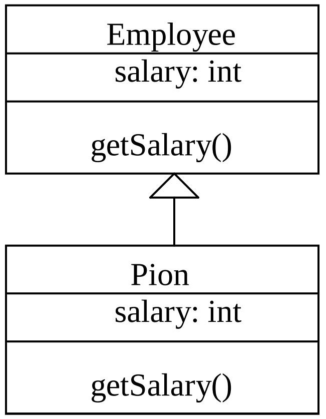
Pure virtual members can be called from the abstract base class

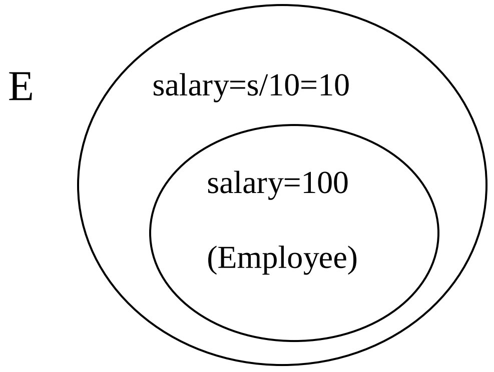
We can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

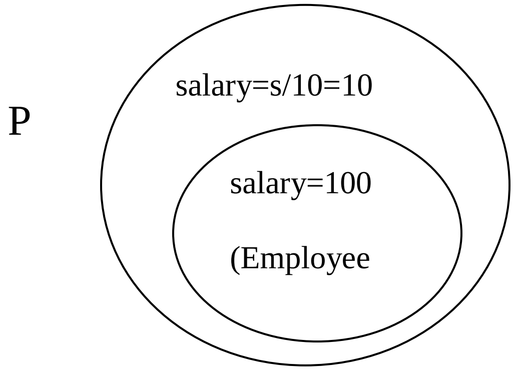
```
// pure virtual members can be called
// from the abstract base class
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
       { width=a; height=b; }
    virtual int area () =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };
```

```
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };
class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

```
// poly_varsmeths
class Employee {
   public int salary;
   public Employee(int s) { salary = s; }
   public int getSalary() { return salary; }
}
class Pion extends Employee {
   public int salary;
   public Pion(int s) { super(s); salary = s/10; }
   public int getSalary() { return salary; }
}

public class poly_varsmeths {
   public static void main(String[] args) {
         Employee E = new Pion(100);
         Pion P = new Pion(100);

         System.out.println(E.salary);
         System.out.println(E.getSalary());
         System.out.println(P.salary);
         System.out.println(P.getSalary());     }}
```







```
abstract class Data {
    public int k;
    public void print1() {
            System.out.println(k);
    }
    abstract public void print2();
}

class Data1 extends Data {
    public Data1(int k) {
            this.k = k;
    }
    public void print2() {
            System.out.println(k*k);
    }
}

class Data2 extends Data {

    public Data2(int k) {
            this.k = k;
    }
    public void print2() {
            System.out.println(k+k);
    }
}
```

```
public class abstract1 {
    public static void main(String args[]) {

            Data1 d1 = new Data1(3);
            Data2 d2 = new Data2(3);
            d1.print1();
            d1.print2();
            d2.print1();
            d2.print2();
    }
}
```