Inheritance

C# Programming

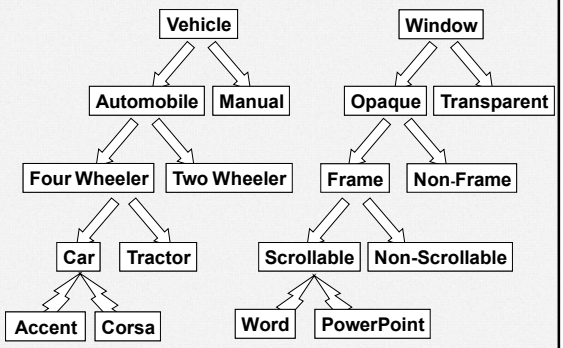
Defining the Pillars of OOP

- All object-based languages (C#, Java, C++, Smalltalk, Visual Basic, etc.) must contend with three core principals, often called the pillars of object-oriented programming (OOP):
  - **Encapsulation:** How does this language hide an object's internal implementation details and preserve data integrity?
  - **Inheritance:** How does this language promote code reuse?
  - **Polymorphism:** How does this language let you treat related objects in a similar way?

The Role of Inheritance

- Inheritance allows you to build new class definitions based on existing class definitions.
- In other words, inheritance allows you to extend the behaviour of a base (or parent) class by inheriting core functionality into the derived class (also called a child class).
- When you establish "is-a" relationships between classes, you are building a dependency between two or more class types. The basic idea behind classical inheritance is that new classes can be created using existing classes as a starting point.

Inheritance



What is an Inheritance?

- Inheritance is a mechanism in which one object acquires properties of another object.
- Inheritance permits reusability of already existing code.
- It is something that we very frequently see in real life such as
  - Car is a vehicle
  - Apple is a fruit
  - Table is Furniture
  - Batsman is Cricketer

Inheritance at a glance

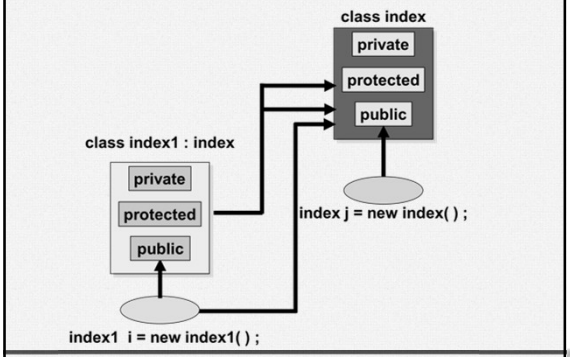
```
class index
{
    protected int count ;
    public index( )
    {
        count = 0 ;
    }
    public void display( )
    {
        C.W ( count ) ;
    }
    public void increment( )
    {
        count += 1 ;
    }
}
```

```
class index1 : index
{
    public void decrement ( )
    {
        count -= 1 ;
    }
}
class Class1
{   static void Main ( string [ ]args )
    {
        index1 i = new index1( ) ;
        i.display( ) ;        0
        i.increment( ) ;
        i.display( ) ;        1
        i.decrement( ) ;
        i.display( ) ;        0
    }
}
```

Presented by

Ranjan Bhatnagar

Access



Points to Ponder

1. Protected members are accessible only within the class and in derived classes.
2. The protected member of parent class cannot be accessed in a child class method using the reference variable of type parent.
3. When an object of child class is created all the data members of the parent class and child class are allocated memory irrespective of their access specifier i.e. public or private or protected.
4. Whenever an object of child class is created, even though the child class constructor is visited first, it's the parent class constructor which is executed first. This is because the child class constructor can then override the code already executed in parent class constructor.
5. Every child class constructor by default first calls default constructor of parent class.

Few More

6. Using "base" in C# and with appropriate arguments a child class constructor can make an explicit call any other constructor of parent class.
7. The order of visiting constructor is child class first and then parent because the child class constructor if required can pass data to the parent class constructor using base.
8. It is recommended that a child class constructor always pass data to the parent class constructor so that the parent class constructor can initialize the data members of the parent class and the child class constructor remains with initializing only its own data members.
9. If the parent class doesn't have default constructor, then every child class constructor must explicitly link to any other constructor of the parent class.
10. The order of execution of destructor is child first and then parent which is reverse order of execution of constructor.

Inheritance Example

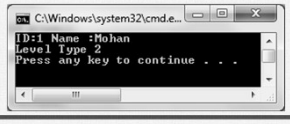
```
using System;
public class Employee
{
    private uint empID = 111111;
    private string empName;

    public string Name
    {
        get { return empName; }
        set
        {
            if (value != null) empName = value;
            else Console.WriteLine("invalid name");
        }
    }
    public uint ID
    {
        get { return empID; }
        set
        {
            if (value != 0) empID = value;
            else Console.WriteLine("invalid ID");
        }
    }
```

Inheritance Example Cont.

```
    public Employee(uint id, string name)
    {
        ID = id;
        Name = name;
    }
    protected void print()
    {
        Console.WriteLine("ID:" + ID + " Name :" + Name);
    }
}
class Manager : Employee
{
   string level;
   static string[] LEVEL = { "Type 1", "Type 2", "Type 3", "Type 4" };
   public string Level
   {
        get { return level; }
        set
        {
            if (Array.BinarySearch(LEVEL, value) >= 0)
                level = value;
            else
                Console.WriteLine("invalid level");
        }
    }
```

Inheritance Example Cont.

```
    public Manager(uint id, string name, string level)
        : base(id, name)
    { Level = level; }
    public new void print()
    {
        base.print();
        Console.WriteLine("Level " + level);
    }
}
class Program
{
    static void Main()
    {
        Manager m = new Manager(1, "Mohan", "Type 2");
        m.print();
    }
}
```



Presented by
Ranjan Bhatnagar

Inheritance and Name Hiding

- It is possible for a derived class to define a member that has the same name as a member in its base class.
- When this happens, the member in the base class is hidden within the derived class.
- C# compiler will issue a warning message.
- This warning alerts you to the fact that a name is being hidden.
- To prevent this warning, the derived class member must be preceded by the new keyword.

Using base to Access a Hidden Name

- There is a keyword base which always refers to the base class of the derived class in which it is used.
- This usage has the following general form:

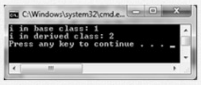
base.member

- Here, member can be either a method or an instance variable.
- This form of base is most applicable to situations in which member names of a derived class hide members by the same name in the base class.

Example Name Hiding

```
using System;
class A
{
    public int i = 0;
}
// Create a derived class.
class B : A
{
    // this i hides the i in A
    new int i;
    public B(int a, int b)
    {
        // this uncovers the i in A
        base.i = a;
        i = b; // i in B
    }
    public void Show()
    {
        // This displays the i in A.
        Console.WriteLine
            ("i in base class: " +
base.i);
        // This displays the i in B.
        Console.WriteLine
            ("i in derived class: " + i);
    }
}
```

```
class NameHiding
{
    static void Main()
    {
        B ob = new B(1,2);
        ob.Show();

    }
}
```



base Keyword

```
using System;
class mybase
{
    int i;
    public mybase(int ii)
    {
        i = ii;
        Console.WriteLine("Base");
    }
}
```

```
class myderv : mybase
{
    public myderv ( int ii )
        :base ( ii )
    {
        Console.WriteLine("Derived");
    }
}
```

```
class Class1
{
    static void Main
                (string[] args)
    {
        myderv d =
            new myderv(10);
    }
}
```

Description

- We have provided a one-argument constructor in the mybase class.
- Then we have derived a class myderv from mybase which also has a one-argument constructor.
- When the object of the derived class is created, the control lands into the one-argument constructor of the derived class from where the default zero-argument constructor of the base class gets called but since we have not provided a zero argument constructor in the base class it would result in an error.

Description

- Thus if we provide a one-argument constructor within a base class then the responsibility of adding a zero-argument constructor also lies with us.
- In the above slide we have shown a way by which we can call the base class's one-argument constructor i.e. One-argument constructor of the mybase class.
- This is done with the use of the base keyword followed by parenthesis.
- In the parenthesis we have passed an argument for the one-argument constructor.

Presented by

Ranjan Bhatnagar

How it Flows?

```
using System;
class mybase
{
    protected int i;
    public mybase()
    {
        i = 4;
    }
}
class myderv : mybase
{
    int j;
    public myderv()
    {
        j = i * 4;
    }
}
```

```
class Class1
{
    static void Main(string[] args)
    {
        myderv d = new myderv();
    }
}
```

Base First
Or
Derived First?

Why Base then Derived?

- Here the class mybase has a protected data member i.
- Next we have derived a class called myderv from mybase.
- This derived class has a data member j. To calculate the value of j, we need the value of i.
- Hence i needs to be constructed before j. i.e the constructor of the base class should be called before the constructor of the derived class.
- This works because by default the zero-argument constructor is called first.
- This scenario is very common where we use the data of the base class in the derived class. Hence this rule becomes useful.

Preventing Inheritance

- The sealed modifier is used to prevent derivation from a class. It is an error if we try to derive a class from a sealed class.

```
using System;
class b
{
    int i;
    public void display()
    {
        Console.WriteLine(i);
    }
}
```

Error

```
class x : b
{
    int j;
    public void display()
    {
        Console.WriteLine(j);
    }
}
```

Multiple Base Classes

- C# demands that a given class have exactly one direct base class.
- You can not create a class type that directly derives from two or more base classes.
- That means C# doesn't support multiple inheritance as was in C++.

```
// Illegal! The .NET platform does not allow
// multiple inheritance for classes!
class WontWork : BaseClassOne, BaseClassTwo
{
    //some definition
}
```

- While a class can have only one direct base class, it is permissible for an interface to directly derive from multiple interfaces.

The "has-a" Relationship

- Code reusability comes in two flavours:
  - Inheritance (the "is-a" relationship)
  - Containment/Delegation model (the "has-a" relationship)
- Just for an example consider Employee class and Manager class, which has "is-a" relationship.
- On the other hand consider SalesCommission class where system required "has-a" relation.
- To maintain "has-a" relationship, you can update the Employee class definition as follows:

```
class Employee
{
    // Contain a SalesCommission object.
    protected SalesCommission empComm = new SalesCommission();
    ...
}
```

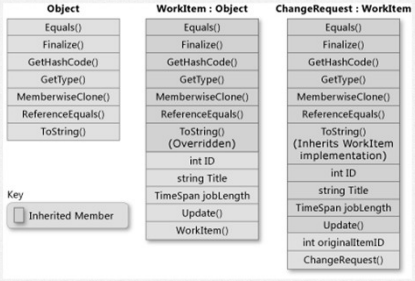
Every Class is a Derived Class

- System.Object is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.
- Languages typically do not require a class to declare inheritance from Object because the inheritance is implicit.
- Because all classes in the .NET Framework are derived from Object, every method defined in the Object class is available in all objects in the system. Derived classes can and do override some of these methods, including: Equals, Finalize, GetHashCode and ToString.

Presented by
Ranjan Bhatnagar

Implicit Inheritance by C#

- Following diagram illustrate how implicit inheritance is performed:



Casting References

- A derived class object can be automatically converted into base class object→ implicit conversion.
- Vice versa requires explicit casting.
- Example
  - Employee t= new Manager(1,"Mohan", "Type 2");
    t.Level = Manager.LEVEL[0];// error
    Manager h=(Manager) t;
    m.Level = Manager.LEVEL[0]; // ok
  - Employee e = new Employee(1, "Mohan");
    Manager m = (Manager)e;
    m.Level = Manager.LEVEL[0];
    //System.InvalidCastException at runtime
- Casting objects when they are not related by inheritance will cause compiler error (unless custom casting is used)

  Manager e = new Manager(1, "Mohan", "Type 2");
  Base c = (Base)e; // error

Protected Method Access and Cast

- A protected method of base class cannot be accessed using the base class instance from the derived class.

```
public class Employee
{
    …
    protected void print()
    {
        Console.WriteLine("ID:" + ID + " Name :" + Name);
    }
}
class Manager : Employee
{
    static void Main()
    {
        Employee e = new Employee(1, "Mohan");
        e.print(); // error
    }
}
```

Custom Casting

- Overloading the cast operator allows the custom casting.
- The keyword implicit or explicit indicate whether casting have to be implicit or explicit.
- The explicit must be specified if there are any values in the class attributes for which the cast will fail or if there is any risk of an exception being thrown.
- They are static methods of the class
- Syntax:

  public static [implicit , explicit] operator return-type (parameter)

Example: Custom casting

- Code provides custom cast methods to convert the Money object into double implicitly and double to Money object explicitly.

```
using System;
class Money
{
    uint rupees;
    uint paise;
    public Money(uint rupees, uint paise)
    {
        this.rupees = rupees;
        this.paise = paise;
    }
    public static implicit operator double(Money m)
    {
        return m.rupees + (m.paise / 100.0);
    }
```

Example Cont.

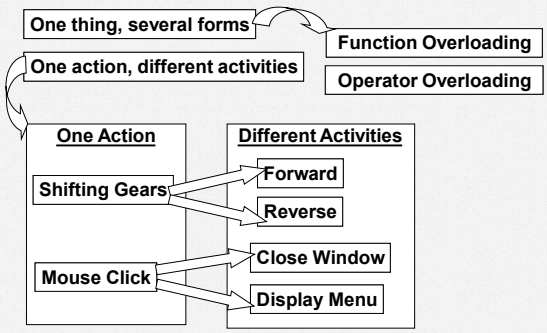
```
public void display()
    {
        Console.WriteLine("Rs. {0}.{1}", rupees, paise);
    }
    public static explicit operator Money(double d)
    {
        uint r = (uint)d;
        uint p = (uint)((d - r) * 100);
        return new Money(r, p);
    }
    public static void Main()
    {
        Money m = new Money(25, 50);
        double m1 = m;
        Console.WriteLine("double value " + m1);
        double d = 30.75;
        m = (Money)d;
        m.display();
    }
}
```

Presented by
Ranjan Bhatnagar

Polymorphism

- Polymorphism is often referred to as the third pillar of object-oriented programming.
- Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:
  - At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
  - Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Polymorphism



Virtual Methods and Overriding

- A virtual method is a method that is declared as virtual in a base class. Each derived class can have its own version of a virtual method.
- When a virtual method is called through a base class reference, C# determines which version of the method to call based upon the type of the object referred to by the reference—and this determination is made at runtime. Feature is also called as Late Binding.
- When a virtual method is redefined by a derived class, the override modifier is used. Thus, the process of redefining a virtual method inside a derived class is called method overriding.

Example: overriding

```
using System;
public class Employee
{
    …
    public virtual void print()
    {
        Console.WriteLine("ID:" + ID + " Name :" + Name);
    }
}
class Manager : Employee
{
    …
    public override void print()
    {
        base.print();
        Console.WriteLine("Level " + level);
    }
}
```

Overriding Rules

- The overridden method must have the same signature as that of the base class method including return type.
- A non-virtual or static method cannot be overridden.
- The overridden base method must be virtual, abstract, or override.
- An overridden derived class method cannot change the accessibility of the virtual method.
- Both the overridden method and the virtual method must have the same access level modifier.
- Modifiers like new, static, virtual, or abstract are not allowed for the overridden method.
- A property can also be overridden like methods and follows the same rules as that of method.

Virtual Methods

```
using System;
class shapes
{
    public virtual void draw()
    {
        Console.WriteLine("Shapes");
    }
}
class rectangle : shapes
{
    public override void draw()
    {
        Console.WriteLine("Rectangle");
    }
}
class circle : shapes
{
    public override void draw()
    {
        Console.WriteLine("Circle");
    }
}
```

```
class Class1
{
    static void Main
        (string[] args)
    {
        shapes[] p = {
            new circle( ),
            new rectangle( ),
            new circle( ),
            new rectangle( ),
            new circle( ),
           };
  for (int i = 0;
       i < p.Length; i++)
         p[i].draw();
    }
}
```

Presented by
Ranjan Bhatnagar

Overriding Virtual Method in VS

- Visual Studio 2010 has a very helpful feature that you can make use of when overriding a virtual member. If you type the word "override" within the scope of a class type (then hit the spacebar), IntelliSense will automatically display a list of all the overridable members defined in your parent classes as shown below:

Override and Virtual

- Override also makes the method virtual

```
using System;
class X
{
    public virtual void f() { Console.WriteLine("X"); }
}
class Y : X
{
    public override void f() { Console.WriteLine("Y"); }
}
class Z : Y
{
    public override void f() { Console.WriteLine("Z"); }
}
class Test
{
    static void Main()
    {
        X x = new Z();
        x.f();
    }
}
```

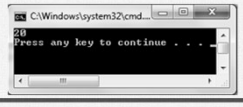
Prints Z

Only Methods can be Overridden

- What happens when the variable is re-declared? Will they also be overridden?
- Only methods can be overridden. For instance the code below prints 20.

```
using System;
class X
{
    int i = 10;
    public virtual void f()
    {
        Console.WriteLine(i);
    }
}
class Y : X
{
    int i = 20;
    public override void f()
    {
        Console.WriteLine(i);
    }
}
```

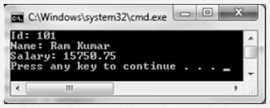
```
class Test
{
    static void Main()
    {
        Y x = new Y();
        x.f();
    }
}
```



Overriding toString()

```
using System;
namespace EmpMgnt
{
  class Employee
  {
    int Id;
    string Name;
    decimal Salary;
    public Employee(int id,
      string name, decimal salary)
    {
      Id = id;
      Name = name;
      Salary = salary;
    }
    public override string ToString()
    {
      string str = "";
      str += "Id: " + Id;
      str += "\nName: " + Name;
      str += "\nSalary: " + Salary;
      return str;
    }
  }
```

```
class Program
{
  static void Main(string[] args)
  {
    Employee e1 = new Employee(101,
      "Ram Kumar", 15750.75M);
    Console.WriteLine(e1.ToString());
  }
}
```



Virtual Members

- When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class.
- The designer of the derived class can choose whether to:
  - override virtual members in the base class,
  - inherit the closest base class method without overriding it
  - define new non-virtual implementation of those members that hide the base class implementations

Example

- A derived class can override a base class member only if the base class member is declared as virtual or abstract.
- The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation.

```
public class BaseClass
{
    public virtual
        void DoWork() { }
    public virtual
        int WorkProperty
    {
        get { return 0; }
    }
}
```

```
public class DerivedClass
                : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Presented by
Ranjan Bhatnagar

More on Virtual Members

- Fields cannot be virtual; only methods, properties, events and indexers can be virtual.
- When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class.

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

- Virtua classes to extend a base class without needing to use the base class implementation of a method.

Sealing Virtual Members

- The sealed keyword can be applied to a class type to prevent other types from extending its behavior via inheritance.
- Sometimes you may not wish to seal an entire class, but simply want to prevent derived types from overriding particular virtual methods.
- For example: For example, assume you do not want parttime salespeople to obtain customized bonuses.
- To prevent the PTSalesPerson class from overriding the virtual GiveBonus() method, you could effectively seal this method in the SalesPerson class as shown in next slide:

SalesPerson and PTSalesPerson Classes

```
// SalesPerson has sealed the GiveBonus() method!
class SalesPerson : Employee
{
    ...
    public override sealed void GiveBonus(float amount)
    {
    ...
    }
}

sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
    float currPay, string ssn, int numbOfSales)
        : base(fullName, age, empID, currPay, ssn, numbOfSales)
    {
    }
    // Compiler error! Can't override this method
    // in the PTSalesPerson class, as it was sealed.
    public override void GiveBonus(float amount)
    {
    }
}
```

The new Modifier

```
using System ;
class b
{
   public virtual void fun( )
   {
      Console.WriteLine("In b");
   }
}
class x : b
{
   public new void fun( )
   {
   Console.WriteLine("In x");
   }
}
```

- Imagine a situation when a method fun( ) in the base class is declared virtual and we have a method with the same name and signature in derived class.
- When we call fun( ) through an upcasted pointer we want that the method fun( ) in the derived class should not override the method fun( ) declared virtual in the base class.
- At such a time we need to declare the method fun( ) in the derived class with the new modifier. The code shown in the slide depicts this.

Thanks

Presented by

Ranjan Bhatnagar