Лекция 33

Полиморфизм

Полиморфизм

Виртуальным называется такой **метод**, который **в базовом классе** объявляется с помощью ключевого слова **virtual**, указываемого перед его именем. Виртуальный метод может быть **переопределен** в одном или нескольких производных классах, т. е. у каждого производного класса может быть свой вариант виртуального метода.

Виртуальный метод переопределяется в производном классе с помощью ключевого слова **override**, указываемого перед его именем.

При обращении к разным типам объектов по ссылке на базовый класс **вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект**.

Полиморфизм

Процесс повторного определения виртуального метода в производном классе называется **переопределением метода**.

При переопределении **имя**, **возвращаемый тип** и **сигнатура** переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется.

Переопределение методов — это еще один способ воплотить в **C#** главный принцип **полиморфизма**: один интерфейс — множество методов.

Переопределение метода реализует **динамический полиморфизм -** механизм разрешения вызова во время выполнения, а не компиляции.

Переопределять виртуальный метод не обязательно. Ведь если в производном классе не предоставляется собственный вариант виртуального метода, то используется его вариант из базового класса.

Пример 1

```
namespace WindowsFormsApp {

  public partial class Form1 : Form {
    Base baseRef;     // ссылка на базовый класс

    private void button3_Click(object sender, EventArgs e)
    {   // Метод Who() в классе Base
      Base baseOb = new Base();
      baseRef = baseOb;
      richTextBox1.AppendText(baseRef.Who());
     //  richTextBox1.AppendText(baseOb.Who());
    }
```

Пример 1

```
    private void button4_Click(object sender, EventArgs e)
    {   // Метод Who() в классе Derived1
        Derived1 dOb1 = new Derived1();
        baseRef = dOb1;
        richTextBox1.AppendText(baseRef.Who());
        // richTextBox1.AppendText(dOb1.Who());
    }
    private void button5_Click(object sender, EventArgs e)
    {  // Метод Who() в классе Base
        Derived2 dOb2 = new Derived2();
        baseRef = dOb2;
        richTextBox1.AppendText(baseRef.Who());
        // richTextBox1.AppendText(dOb2.Who());
    }
}
```

Пример 1

```
class Base {
    public virtual string Who() {
        // Создать виртуальный метод в базовом классе
        return "Метод Who () в классе Base";
    }
}
class Derived1 : Base {
    public override string Who() {
        // Переопределить метод Who() в производном классе
        return "Метод Who() в классе Derived1";
    }
}
class Derived2 : Base {
    // В этом классе метод Who() не переопределяется
}
}
```

Пример 2

```
using System;
class Base {
    public virtual void Who() {
    // Создать виртуальный метод в базовом классе
        Console.WriteLine("Метод Who() в классе Base");
    }
}
class Derived1 : Base {
    public override void Who() {
   // Переопределить метод Who() в производном классе
        Console.WriteLine("Метод Who() в классе
   Derived1");
    }
}
```

Пример 2

```
class Derived2 : Derived1{
     // В этом классе метод Who() не переопределяется
}
class Derived3 : Derived2 {
    // И в этом классе метод Who() не переопределяется
}
class NoOverrideDemo2 {
   static void Main() {
     Derived3 dOb = new Derived3();
     Base baseRef;      // ссылка на базовый класс
     baseRef = dOb;
     baseRef.Who();  // вызов метода Who() из класса Derived1
   }
}
```

Пример 3

```
using System;
class TwoDShape {
      double pri_width;
      double pri_height;
      public TwoDShape() { // Конструктор по умолчанию
          Width = Height = 0.0;
           name = "null";
     }
     public TwoDShape(double w, double h, string n) {
     // Параметризированный конструктор
           Width = w;
           Height = h;
           name = n;
      }
```

Пример 3

```
    public TwoDShape(double x, string n) {
// Сконструировать объект равной ширины и высоты
        Width = Height = x;
        name = n;
    }
    public TwoDShape(TwoDShape ob) {
// Сконструировать копию объекта TwoDShape
        Width = ob.Width;
        Height = ob.Height;
        name = ob.name;
    }
```

Пример 3

```
 public double Width { // Свойства ширины и высоты объекта
      get { return pri_width; }
      set { pri_width = value < 0 ? -value : value; }
  }
  public double Height {
      get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }
   public string name { get; set; }
   public void ShowDim() { Console.WriteLine("Ширина и высота
  равны " + Width + " и " + Height);  }
   public virtual double Area() {
       Console.WriteLine("Метод Area() должен быть
  переопределен");
       return 0.0;
    }
}
```

Пример 3

```
class Triangle : TwoDShape {   // Класс для треугольников
    string Style;
    public Triangle() { Style = "null"; }
    public Triangle(string s, double w, double h) : base(w, h,
   "треугольник") {
        Style = s;
    }
    public Triangle(double x) : base(x, "треугольник") {
        Style = "равнобедренный";
    }
    public Triangle(Triangle ob) : base(ob) { Style = ob.Style; }
    public override double Area() { return Width * Height / 2; }
    public void ShowStyle() {
        Console.WriteLine("Треугольник " + Style);
    }
}
```

Пример 3

```
class Rectangle : TwoDShape {
    // Конструктор для класса Rectangle
    public Rectangle(double w, double h) : base(w, h,
  "прямоугольник"){ }
    // Сконструировать квадрат
    public Rectangle(double x) : base(x, "прямоугольник") { }
    // Сконструировать копию объекта типа Rectangle
    public Rectangle(Rectangle ob) : base(ob) { }
    // Возвратить логическое значение true,
    // если прямоугольник окажется квадратом
    public bool IsSquare() {
        if (Width == Height) return true;
        return false;
    }
    // Переопределить метод Area() для класса Rectangle
    public override double Area() { return Width * Height; }
}
```

Пример 3

```
class DynShapes {
   static void Main() {
     TwoDShape[] shapes = new TwoDShape[5];
     shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
     shapes[1] = new Rectangle(10);
     shapes[2] = new Rectangle(10, 4);
     shapes[3] = new Triangle(7.0);
     shapes[4] = new TwoDShape(10, 20, "общая форма");
     for (int i=0; i < shapes.Length; i++) {
        Console.WriteLine("Объект — " + shapes[i].name);
        Console.WriteLine("Площадь равна " + shapes[i].Area());
        Console.WriteLine();
     }
}
```

Полиморфизм

В **C#** имеется возможность предотвратить наследование класса с помощью ключевого слова **sealed**.

Класс не допускается объявлять одновременно как **abstract** и **sealed**, поскольку сам абстрактный класс реализован не полностью и опирается в этом отношении на свои производные классы, обеспечивающие полную реализацию.

Ниже приведен пример объявления класса типа **sealed**.

```
sealed class A {
    // ...
}
class B : A { // ОШИБКА! Наследовать класс A нельзя
   // ...
}
```

Ключевое слово **sealed** может быть также использовано в виртуальных методах для предотвращения их дальнейшего переопределения.

Пример 5

```
class B {
        public virtual void MyMethod() { /* ... */ }
}
class D : B {
   // Здесь герметизируется метод MyMethod() и
   // предотвращается его дальнейшее переопределение
        sealed public override void MyMethod() { /* ... */ }
}
class X : D {
   // Ошибка! Метод MyMethod() герметизирован!
        public override void MyMethod() { /* ... */ }
}
```

Класс **object**

В **C#** предусмотрен специальный класс **object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от **object**. Это, в частности, означает, что переменная ссылочного типа **object** может ссылаться на объект любого другого типа. Кроме того, переменная типа **object** может ссылаться на любой массив, поскольку в **C#** массивы реализуются как объекты. Формально имя **object** считается в **C#** еще одним обозначением класса **System.Object**, входящего в библиотеку классов для среды .NET Framework.

Класс **object**

Метод **Назначение**

public virtual bool **Equals** (object ob) - Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке ob

public static bool **Equals** (object obj A, object objB) - Определяет, является ли объект, доступный по ссылке objA, таким же, как и объект, доступный по ссылке objB

protected **Finalize** () - Выполняет завершающие действия перед "сборкой мусора". В C# метод Finalize () доступен посредством деструктора

public virtual int **GetHashCode** () - Возвращает хеш-код, связанный с вызывающим объектом

Класс **object**

Метод **Назначение**

public Type **GetType** () - Получает тип объекта во время выполнения программы

protected object **MemberwiseClone** () - Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены

public static bool **ReferenceEquals** (obj objA, object objB) - Определяет, делаются ли ссылки objA и objB на один и тот же объект

public virtual string **ToString**() - Возвращает строку, которая описывает объект

Пример 6

```
class MyClass {
    static int count = 0;
    int id;
    public MyClass () {
        id = count;
        count++;
    }
    public override string ToString() {
        return "Объект #" + id + " типа MyClass";
    }
}
```

Пример 6

```
class Test {
    static void Main() {
        MyClass ob1 = new MyClass();
        MyClass ob2 = new MyClass();
        MyClass ob3 = new MyClass();
        Console.WriteLine(ob1);
        Console.WriteLine(ob2);
        Console.WriteLine(ob3);
    }
}
```

Класс **object**

Все типы в **C#**, включая и простые типы значений, являются производными от класса **object**. Следовательно, ссылкой типа **object** можно воспользоваться для обращения к любому другому типу, в том числе и к типам значений.

Когда ссылка на объект класса **object** используется для обращения к типу значения, то такой процесс называется **упаковкой**. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект. Упаковка происходит автоматически. Для этого достаточно присвоить значение переменной ссылочного типа **object**.

Распаковка представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки на объект класса **object** к соответствующему типу значения. Попытка распаковать объект в другой тип может привести к ошибке.

Пример 7

```
class BoxingDemo {
   static void Main() {
     int x;
     object obj;
     x = 10;
     obj = x; // упаковать значение переменной x в объект
     int y = (int)obj; // распаковать значение из объекта,
    // доступного по ссылке obj, в переменную типа int
     Console.WriteLine(y);
  }
}
```

Пример 8

```
class BoxingDemo {
   static void Main() {
      int x;
      x = 10;
      Console.WriteLine("Значение x равно: " + x);
      // значение переменной x автоматически упаковывается
      // когда оно передается методу Sqr()
      x = BoxingDemo.Sqr(x);
      Console.WriteLine("Значение x в квадрате равно: " + x);
   }
   static int Sqr(object o) {
      return (int)o * (int)o;
  }
}
```

Пример 9

```
// Благодаря упаковке становится возможным
// вызов методов по значению!
using System;
class MethOnValue {
    static void Main() {
        Console.WriteLine(10.ToString()) ;
    }
}
```

Пример 10

```
class GenericDemo {
    static void Main() {
      object[] ga = new object[10];
       // Сохранить целые значения
      for (int i=0; i < 3; i++) ga[i] = i;
       // Сохранить значения типа double
      for (int i=3; i < 6; i++) ga[i] = (double) i / 2;
      // Сохранить две строки, а также значения типа  bool и char
       ga [6] = "Привет";        ga [7] = true;
       ga [8] = 'X';                  ga [9] = "Конец";
       for (int i = 0; i < ga.Length; i++)
          Console.WriteLine("ga [" + i +"]:"+ ga [i] +"");
   }
}
```

Пример 10

Выполнение этой программы приводит к следующему результату:

ga [0] : 0

ga [1] : 1

ga[2] : 2

ga[3]: 1.5

ga[4] : 2

ga[5]: 2.5

ga[6]: Привет _

ga[7]: True

ga[8] : X

ga[9]: Конец

Класс **object**

Несмотря на то что универсальный характер класса **object** может быть довольно эффективно использован в некоторых ситуациях, было бы ошибкой думать, что с помощью этого класса стоит пытаться обойти строго соблюдаемый в **C#** контроль типов: целое значение следует хранить в переменной типа **int**, строку — в переменной ссылочного типа **string** и т.д.

Внедрение в **C# обобщений** позволило без труда определять классы и алгоритмы, автоматически обрабатывающие данные разных типов, соблюдая типовую безопасность. Благодаря обобщениям отпала необходимость пользоваться классом **object** как универсальным типом данных при создании нового кода.

Контрольные вопросы

1. Каким образом реализуется динамический полиморфизм?
2. Каково назначение абстрактных методов и классов?
3. Какими возможностями обладает класс object?

Виртуальные, переопределяющие и абстрактные методы

Если объявление метода экземпляра содержит модификатор **virtual**, метод является **виртуальным методом**.

Виртуальный метод может быть **переопределен** в производном классе. Если объявление метода экземпляра содержит модификатор **override**, метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Объявление виртуального метода определяет новый метод. Объявление переопределяющего метода уточняет существующий виртуальный метод, предоставляя его новую реализацию.

Виртуальные, переопределяющие и абстрактные методы

Вместе с ключевым словом **override** может также использоваться и ключевое слово **sealed** (герметизированный), указывающее, что в данный метод больше не могут вноситься изменения ни в каких производных классах, т.е. метод не может **переопределяться** в производных классах.

```
public class MyDerivedClass : MyBaseClass {
        public override sealed void DoSomething () {
          // Реализация в производном классе,
          // переопределяющая базовую реализацию.
        }
}
```

Виртуальные, переопределяющие и абстрактные методы

Абстрактным называется виртуальный метод без реализации.

Объявление абстрактного метода осуществляется с использованием модификатора **abstract** и допускается только в классе, объявленном как **abstract**.

В каждом неабстрактном производном классе необходимо переопределять абстрактный метод.

Пример 16

```
public abstract class Expression{
   public abstract double Evaluate(Hashtable vars);
}
public class Constant: Expression{
   double value;
   public Constant(double value) {
    this.value = value;
   }
  public override double Evaluate(Hashtable vars){
    return value;
   }
}
```

Пример 16

```
public class VariableReference: Expression{
   string name;
   public VariableReference(string name) {
    this.name = name;
   }
   public override double Evaluate(Hashtable vars){
    object value = vars[name];
    if (value == null)  throw new Exception("Не
   определена переменная: " + name);
    return Convert.ToDouble(value);
   }
}
```

Пример 16

```
public class Operation: Expression{
    Expression left;
    char op;
    Expression right;
   public Operation(Expression left, char op,
   Expression right) {
    this.left = left;
    this.op = op;
    this.right = right;
   }
```

Пример 16

```
public override double Evaluate(Hashtable vars){
 double x = left.Evaluate(vars);
 double y = right.Evaluate(vars);
 switch (op) {
     case '+': return x + y;
     case '-': return x - y;
     case '*': return x * y;
     case '/': return x / y;
 }
 throw new Exception("Не определен оператор");
}
}
```

Пример 16

Четыре приведенных выше класса могут использоваться для моделирования арифметических выражений. Например, с помощью экземпляров этих классов выражение x + 3 можно представить следующим образом.

```
Expression e = new Operation(new
  VariableReference("x"), '+', new
  Constant(3));
```

Пример 16

```
class Test{ //классы Expression используются для вычисления
//  выражения x * (y + 2) с различными значениями x и y
    static void Main() {
      Expression e = new Operation(
     new VariableReference("x"),
     '*',
     new Operation(
          new VariableReference("y"),
          '+',
          new Constant(2)
      )
    );
```

Пример 16

```
    Hashtable vars = new Hashtable();
    vars["x"] = 3;
    vars["y"] = 5;
    Console.WriteLine(e.Evaluate(vars)); // Вывод "21"
    vars["x"] = 1.5;
    vars["y"] = 9;
    Console.WriteLine(e.Evaluate(vars)); //Вывод "16.5"
    }
}
```