Лекция 3

Определение

- Полиморфи́зм — возможность объектов с **одинаковой** спецификацией иметь **различную** реализацию.

Виды полиморфизма

- Ad hoc полиморфизм (специализированный полиморфизм)
  - Перегрузка функций (методов)
  - Перегрузка операторов
- Полиморфизм подтипов
  - Полиморфизм включения
- Параметрический полиморфизм
  - Параметрические методы
  - Параметрические типы

Ad hoc полиморфизм

- Ad hoc полиморфизм – это вид полиморфизма , при котором полиморфные методы (функции) могут применяться с **различными типами данных**.

Перегрузка функций

- Создадим класс, а в нем – простой метод для сложения 2-х целочисленных чисел.

```
class overloadMethods
  {
     int methodSum(int a, int b)
     {
        return a + b;
     }
  }
```

Перегрузка функций

- Теперь создадим в этом классе еще один метод для сложения, но уже вещественных чисел

```
double methodSum(double a, double b)
    {
       return a + b;
    }
```

Перегрузка функций

- И никакой ошибки

```
class overloadMethods
{
    0 references
    int methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b, int c)
    {
        return (double)a + b;
    }
```

Как делать нельзя

- А здесь есть ошибка.
- В чем она заключается?

```
class overloadMethods
{
    0 references
    int methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b, int c)
    {
        return (double)a + b;
    }
```

Как оно работает

- У нас есть набор функций имеющих одинаковое имя, но разный набор принимаемых параметров.
- При вызове функции мы должны **однозначно** определить, какую функцию вызывать.

Как оно работает

- Однозначно означает, что в этом фрагменте кода при **каждом** запуске будет вызываться одна из функций и только она. Не зависимо от значений, передаваемых в функции при каждом вызове.

Как делать нельзя

- Возвращаемся к ошибке. Что же здесь не так?

```
class overloadMethods
{
    0 references
    int methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b, int c)
    {
        return (double)a + b;
    }
```

В чем причина

- При вызове функции, ее можно однозначно идентифицировать ее **только** по передаваемым в нее параметрам (при условии наличия функций с одинаковыми именами), но **невозможно различать** функции по типу возвращаемого значения (потому что компилятору этого мало при определении однозначности)

Как делать можно

- Функция может различаться по следующим признакам:
  - Разный тип передаваемых параметров
  - Разное количество передаваемых параметров
  - Комбинация первых двух случаев

Пример

- Возможные варианты:

```
class overloadMethods
{
    0 references
    int methodSum(int a, int b)
    {
        return a + b;
    }
    0 references
    double methodSum(int a, int b, int c)
    {
        return (double)a + b;
    }
    0 references
    double methodSum(double a, double b)
    {
        return a + b;
    }
    0 references
    double methodSum(double a, int b)
    {
        return a + b;
    }
```

Перегрузка конструкторов

- Конструкторы класса – это методы, следовательно принципы перегрузки функций применимы и к ним.

```
class overloadMethods
{
    0 references
    public overloadMethods()
    { }

    0 references
    public overloadMethods(int a)
    { }

    0 references
    public overloadMethods(int a, int b)
    { }

    0 references
    public overloadMethods(double d)
    { }
```

Перегрузка операций

- В C#, подобно любому языку программирования, имеется готовый набор операций, используемых для выполнения базовых операций над **встроенными** типами.

Виды операций

- Математические операции
  - +, -, *, /
- Логические операции над числами
  - &, |
- Операции сдвига
  - <<, >>
- Операции сравнения
  - ==, !=, <, >, <=, >=
- Операции присваивания
  - =, +=, -=

Перегрузка операций

- Перегрузка операций позволяет **задать смысл** стандартных операций C#, таких как сложение, вычитание, инкремент, декремент и т.д., для **классов**, определяемых пользователем.

Как перегружать

- Перегрузка операций строится на основе открытых статических методов, объявляемых с использованием ключевого слова **operator**.
- Синтаксис перегрузки :

public static <тип возвращаемого значения>
operator <операция>(<параметры>)

Пример

- В классе определим целочисленное поле и переопределим оператор сложения:

```
class overloadMethods
{
    int field;
    0 references
    public static overloadMethods operator +(overloadMethods a, overloadMethods b)
    {
        a.field += b.field;
        return a;
    }
```

Что и как можно перегружать

- В качестве возвращаемого значения может выступать **любой** тип данных. Все будет зависеть от **логики**, которую закладывают в перегружаемую операцию.

Что и как можно перегружать

- Типы параметров, передаваемые при перегрузке операций тоже могут быть **любыми** и так же зависят исключительно от **логики** перегрузки.

Что и как можно перегружать

- **Количество** передаваемых в функцию параметров зависит от перегружаемой **операции**.
- Операции бывают:
  - Унарные
  - Бинарные
  - тернарные

Примеры

```
class overloadMethods
{
    int field;
    0 references
    public static overloadMethods operator +(overloadMethods a, overloadMethods b)
    {
        a.field += b.field;
        return a;
    }
    0 references
    public static overloadMethods operator +(overloadMethods a, int b)
    {
        a.field += b;
        return a;
    }
    0 references
    public static overloadMethods operator ++(overloadMethods a)
    {
        a.field++;
        return a;
    }
```

Правила перегрузки операций

Операторы	Возможность перегрузки
+, -, !, ~, ++, --, true, false	Эти унарные операторы **можно** перегрузить.
+, -, *, /, %, &, |, ^, <<, >>	Эти бинарные операторы **можно** перегрузить.
==, !=, <, >, <=, >=	Операторы сравнения **могут** быть перегружены*.
&&, ||	Условные логические операторы **не могут быть** перегружены, но они оцениваются с помощью **&** и **|**, которые могут быть перегружены.
[]	Оператор индексирования массива **не может быть** перегружен, но можно определить индексаторы.

Правила перегрузки операций

Операторы	Возможность перегрузки
(T)x	Оператор приведения типов **не может быть** перегружен, но можно определить новые операторы преобразования (см. explicit и implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Операторы присваивания нельзя перегрузить, но **+=**, например, вычисляется с помощью **+**, который перегрузить можно.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	Эти операторы не могут быть перегружены.

Примечание

- При перегрузке операторов сравнения они должны перегружаться **парами**;
- то есть если оператор == перегружается, оператор != тоже должен перегружаться.
- Обратное также верно, и сказанное относится также к парам операторов < и >, <= и >=.

Модификатор static

- Модификатор static используется для объявления статического члена, **принадлежащего** собственно **типу**, а **не** конкретному **объекту**.

С чем его можно «есть»

- Модификатор static **можно** использовать с классами, полями, методами, свойствами, операторами, событиями и конструкторами.
- **Нельзя** — с индексаторами, деструкторами или типами, отличными от классов.

Статический класс

- **Нельзя** создавать экземпляры статического класса (нельзя использовать ключевое слово new для создания переменной типа класса)
- Поскольку нет переменной экземпляра, **доступ** к членам статического класса осуществляется с использованием самого **имени класса**.

Пример

- Создаем статичный класс со статичными членами класса

```
static class ClassStatic
{
    public static int field1;

    0 references
    public static void method()
    { }
}
```

```
class UseClass
{
    0 references
    void method1()
    {
        ClassStatic.field1 = 45;

        ClassStatic.method();
    }
```

- Как к нему обращаться

Ранее связывание

- Связывание – это связь между именем вызываемого метода и непосредственно местом, где этот метод реализован.
- Перегрузка методов и конструкторов относится к раннему связыванию. Т.е. когда компилятор на этапе компиляции, исходя из имеющихся типов данных, передаваемых в вызываемый метод, определяет что вызывать.

Ранее связывание

- Преимущество раннего связывания в том, что при невозможности установить связь между вызываемым именем и существующими методами, будет выдана ошибка на этапе компиляции, и сборка проекта прекращается.

Позднее связывание

- Выбор переопределённого метода откладывается на момент обращения к методу во время исполнения программы. На момент компиляции компилятор точно не знает, какой из группы переопределённых методов будет выполнен.

Переопределение виртуального метода

- Выбор переопределённого метода откладывается на **момент выполнения**

```
class ClassParent
{
    0 references
    public virtual void method()
    {
        y += x;
    }
```

```
3 references
class ClassChild : ClassParent
{
    1 reference
    public override void method()
    {
        //base.method(); // это создалось по умолчанию
        y += 40;
    }
```

Полиморфизм подтипов

- Этот вид полиморфизма заключается в том, что вызывающий код использует объект, опираясь только на его **представление** (интерфейс), не зная при этом **фактического** типа.

Пример

- Опять вернемся к предыдущей лекции. Есть интерфейс и есть его наследник(и)

```
interface Inter1
{
    1 reference
    void method1();

    1 reference
    void method2(int x, double y);

    1 reference
    int method3();
}
```

```
class childInter : Inter1
{
```

```
class childInter2 : Inter1
{
```

Пример

- И есть метод, который принимает переменную типа интерфейс и вызывает его метод

```
class ClassMain
{
    2 references
    public void method(Inter1 ert)
    {
        ert.method1();
    }
```

- На самом деле в метод будет передаваться переменная-наследник от этого интерфейса.

Пример

- В зависимости от того, как каждый из наследников переопределил что делать в method1, будут выполняться разные действия при вызове одного и того же метода

```
class simpleClass
{
    0 references
    void method()
    {
        ClassMain main = new ClassMain();
        main.method(new childInter());
        main.method(new childInter2());
    }
```

И еще пример

```
public abstract class Animal
{
    public abstract String Talk();
}

public class Cat : Animal
{
    public override String Talk()
    {
        return "Meow!";
    }
}

public class Dog : Animal
{
    public override String Talk()
    {
        return "Woof!";
    }
}
```

```
public class Program
{
    private static void Write(Animal animal)
    {
        Console.WriteLine(animal.Talk());
    }

    public static void Main(String args[])
    {
        Write(new Cat());
        Write(new Dog());
    }
}
```

Параметрический полиморфизм

- Параметрический полиморфизм позволяет определить функцию или тип данных обобщённо, так что значения обрабатываются идентично **вне зависимости от их типа**.

Параметрический полиморфизм

- Параметрическая полиморфная функция использует аргументы на основе **поведения**, а не значения, апеллируя лишь к необходимым ей **свойствам аргументов**, что делает её применимой в любом контексте, где **тип** объекта **удовлетворяет** заданным **требованиям** поведения.

Пример

- Параметрический класс, который не зависит от типа данных

```
class ClassPolimorth<T>
{
    1 reference
    public void Swap(ref T a, ref T b)
    {
        T temp = a;
        a = b;
        b = temp;
    }

}
```

```
class UseClass
{
    0 references
    void method()
    {
        int a = 3;
        int b = 5;
        ClassPolimorth<int> cl = new ClassPolimorth<int>();
        cl.Swap(ref a, ref b);
    }
}
```

Как его задавать

- После имени класса в угловых скобках ‘<‘, ‘>’ указывается имя параметра, вместо которого будет подставляться конкретный тип данных.

Возможности

- Типов-параметров в классе может быть несколько

```
class ClassPolimorth<T, U, X>
{
    U filed1;
    X field2;
    0 references
    public void Swap(ref T a, ref T b)
    {
        T temp = a;
        a = b;
        b = temp;
    }

    0 references
    public void Save(X a, U b)
    {
        filed1 = b;
        field2 = a;
    }
}
```

Как оно работает

- При создании объекта от такого класса, по сути, происходит создание нового класса, где вместо типа-параметра (например, T) подставляется конкретный тип данных.

```
class UseClass
{
    0 references
    void method()
    {
        int a = 3;
        int b = 5;
        ClassPolimorth<int, char, int> cl = new ClassPolimorth<int, char, int>();
        cl.Swap(ref a, ref b);
        cl.Save(34, 'r');
    }
}
```