Урок 3

Объектно-ориентированное программирование. Часть 3

Обобщения. Делегаты и события. Паттерн «наблюдатель».

Обобщения

Обобщенное программирование (generic programming) – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само описание. Продемонстрируем пример необобщенной реализации функции обмена значениями двух переменных **Swap** с использованием перегруженных методов. Используя перегруженные методы, в C# мы можем создавать функции с одинаковыми именами, но с разными параметрами:

```
static void Swap(ref int A,ref  int B)
{
     int t;
     t = A;
     A = B;
     B = t;
}
static void Swap(ref  double A, ref  double B)
{
     double t;
     t = A;
     A = B;
     B = t;
}
static void Swap(ref object a,ref object b)
{
    object t = a;
    a = b;
    b = t;
}
```

Пример реализации **Swap** при помощи обобщения:

```
static void Swap<T>(ref  T A, ref  T B) // Обобщенные методы не могут иметь
out параметры
{
     T t;
     t = A;
     A = B;
     B = t;
}
```

После появления первого выпуска платформы **.NET** программисты часто использовали пространство имен **System.Collections**, чтобы получить более гибкий способ управления данными в приложениях. Но с версии .NET 2.0 язык программирования C# был расширен поддержкой обобщения (generic). Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имен, связанным с коллекциями – **System.Collections.Generic**.

Термин «обобщение», по сути, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра.

С помощью обобщений можно создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризованным типом данных, называется обобщенным (обобщенный класс или метод).

В C# всегда была возможность создавать обобщенный код, оперируя ссылками типа **object**. А поскольку класс **object** является базовым для всех остальных классов, по ссылке типа **object** можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа **object**.

В таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа **object** в конкретный тип данных требовалось приведение типов. Это потенциальный источник ошибок: приведение типов могло быть неумышленно выполнено неверно. Обобщения решили проблему типовой безопасности. Они упростили и процесс в целом, поскольку исключили необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных. Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

Использование обобщений

```
public class Animals
{
	public object Gender;
}

public enum Sex
{
	Male,
	Female,
	Undefined
}

public class Program
{
	private static void Main()
	{
		Animals animals = new Animals();
		animals.Gender = "Male"; // Можем назначить пол строкой
		animals.Gender = Sex.Male; // Можем назначить пол перечислением
	}
}
```

Такое решение – не из оптимальных. Дело в том, что в данном случае мы сталкиваемся с такими явлениями, как упаковка (boxing) и распаковка (unboxing). Как мы уже упоминали, упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа **int**) к типу **object**. При упаковке общеязыковая среда **CLR** обертывает значение в объект типа **System**.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing) преобразует объект типа **object** к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо выполнять преобразования. Кроме того, существует проблема безопасности типов.

```
public class Animals<T>
{
	public T Gender;
}

public enum Sex
{
	Male,
	Female,
	Undefined
}

public class Program
{
	private static void Main()
	{
		Animals<Sex> animals = new Animals<Sex>
		{
			Gender = Sex.Male
		};
		Animals<string> animals1 = new Animals<string>
		{
			Gender = "Male"
		};
	}
}
```

Используя букву **T** в описании class **Animals<T>**, мы указываем, что данный тип будет использоваться этим классом. В классе мы создаем поле данного типа. Причем сейчас нам неизвестно, что это будет за тип – и он может быть любым. А параметр **T** в угловых скобках – это «параметр универсального типа», так как вместо него можно подставить любой тип.

Иногда возникает необходимость присвоить переменным универсальных параметров начальное значение, в том числе и **null**. Но напрямую мы его присвоить не можем. В этом случае надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения **null**, а типам значений – значение **0**:

```
public class Animals<T>
{
    public T Gender = default(T);
}
```

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов:

```
public class Animals<T>
{
    public static T Gender = default(T);
}
```

Для **Animals<string>** и для **Animals<Sex>** будет создана своя переменная **Gender**.

Обобщения могут использовать несколько универсальных параметров, которые могут представлять различные типы, одновременно:

```
public class Animals<T, U>
{
	public T Gender = default(T);
	public U Id = default(U);
}

public class Program
{
	private static void Main()
	{
		Animals<Sex, string> animals = new Animals<Sex, string>
		{
			Gender = Sex.Male,
			Id = "1"
		};
	}
}
```

Ограничения обобщений

Чтобы конкретизировать тип, применяется контекстно-зависимое ключевое слово **where**:

```
public class Animals<T> where T : struct // Вместо T можно подставить только
значимый тип данных
{
	public T Gender = default(T);
}

public enum Sex
{
	Male,
	Female,
	Undefined
}

public class Program
{
	private static void Main()
	{
		Animals<Sex> animals = new Animals<Sex> {Gender = Sex.Male};

		// Animals<string> animals1 = new Animals<string>(); Ошибка
string - ссылочный тип данных
		// animals1.Gender  = "Male";
	}
}
```

```
where T : struct - аргумент типа должен быть структурного типа, кроме Nullable.
where T : class - аргумент типа должен иметь ссылочный тип; это также
распространяется на тип любого класса, интерфейса, делегата или массива.
where T : <base class name> - аргумент типа должен являться или быть
производным от указанного базового класса
where T : U - аргумент типа, поставляемый для T, должен являться или быть
производным от аргумента, поставляемого для U. Это называется неприкрытым
ограничением типа
where T : new () - ограничение указывает, что аргумент любого типа в объявлении
универсального класса должен иметь открытый конструктор без параметров.
Использовать ограничение new можно только в том случае, если тип не является
абстрактным
```

Наследование обобщенных типов

Обобщения также поддерживают различные варианты наследования.

Первый вариант – когда производный класс типизирован тем же типом, что и базовый:

```
namespace GeekBrains
{
	public enum Sex
	{
		Male,
		Female,
		Undefined
	}

	public class Animals<T>
	{
		public T Gender = default(T);

		public Animals(T gender)
		{
			Gender = gender;
		}
	}

	public class Cat<T> : Animals<T>
	{
		public Cat(T gender) : base(gender)
		{
			Gender = gender;
		}
	}

	public class Program
	{
		private static void Main()
		{
			Animals<string> animals = new Animals<string>("M");
			Animals<bool> animals1 = new Cat<bool>(true);
```

```
                Cat<Sex> animals2 = new Cat<Sex>(Sex.Male);
            }
        }
}
```

Второй вариант – создание обычного необобщенного класса. В этом случае при наследовании у базового класса надо явным образом определить используемый тип:

```
namespace GeekBrains
{
        public enum Sex
        {
                Male,
                Female,
                Undefined
        }

        public class Animals<T>
        {
                public T Gender = default(T);

                public Animals(T gender)
                {
                        Gender = gender;
                }
        }

        public class Cat : Animals<Sex>
        {
                public Cat(Sex gender) : base(gender)
                {
                        Gender = gender;
                }
        }

        public class Program
        {
                private static void Main()
                {
                        Cat animals = new Cat(Sex.Male);
                        Animals<Sex> animals1 = new Cat(Sex.Male);
                        Animals<Sex> animals2 = new Animals<Sex>(Sex.Male);
                }
        }
}
```

Третий вариант – когда типизация класса-наследника отличается от универсального параметра в базовом классе. В этом случае для базового класса также надо указать используемый тип:

```
namespace GeekBrains
{
	public enum Sex
	{
		Male,
		Female,
		Undefined
	}

	public class Animals<T>
	{
		public T Gender = default(T);

		public Animals(T gender)
		{
			Gender = gender;
		}
	}

	public class Cat<T> : Animals<Sex>
	{
		public T Id;
		public Cat(Sex gender) : base(gender)
		{
			Gender = gender;
		}
	}

	public class Program
	{
		private static void Main()
		{
			Cat<int> animals = new Cat<int>(Sex.Male);
			Animals<Sex> animals1 = new Cat<int>(Sex.Male);
			Animals<Sex> animals2 = new Animals<Sex>(Sex.Male);
		}
	}

}
```

В производных классах можно сочетать использование универсального параметра из базового класса с применением своих параметров:

```
using System;

namespace GeekBrains
{
	public enum Sex
	{
		Male,
		Female,
		Undefined
	}

	public class Animals<T>
	{
		public T Gender = default(T);

		public Animals(T gender)
		{
			Gender = gender;
		}
	}

	// Аргумент типа, предоставляемый в качестве T, должен совпадать с аргументом, предоставляемым в качестве U, или быть производным от него.
	public class Cat<T, R, U> : Animals<R>
		where T : U
	{
		public T Id;
		public Cat(R gender) : base(gender)
		{
			Gender = gender;
		}
	}

	public class Program
	{
		private static void Main()
		{
			Cat<int, Sex, Object> animals = new Cat<int, Sex, Object>(Sex.Male);
			Animals<Sex> animals1 = new Cat<long, Sex, long>(Sex.Male);
			Animals<Sex> animals2 = new Animals<Sex>(Sex.Male);
		}
	}
}
```

Рассмотрим необобщенную реализацию интерфейса **IComparable**. Для этого в класс **Asteroid** добавим поле **Power** (сколько нужно выстрелов для его разрушения):

```
using System;
using System.Drawing;

namespace MyGame
{
    class Asteroid : BaseObject, ICloneable, IComparable
    {
        public int Power { get; set; } = 3; // Начиная с версии C# 6.0 была
добавлена инициализация автосвойств

        public Asteroid(Point pos, Point dir, Size size) : base(pos, dir, size)
        {
            Power = 1;
        }

        public object Clone()
        {
            // Создаем копию нашего робота
            Asteroid asteroid = new Asteroid(new Point(Pos.X, Pos.Y), new
Point(Dir.X, Dir.Y),
                new Size(Size.Width, Size.Height)) {Power = Power};
            // Не забываем скопировать новому астероиду Power нашего астероида
            return asteroid;
        }
        public override void Draw()
        {
            Game.Buffer.Graphics.FillEllipse(Brushes.White, Pos.X, Pos.Y,
Size.Width, Size.Height);
        }
        int IComparable.CompareTo(object obj)
        {
            if (obj is Asteroid temp)
            {
                if (Power > temp.Power)
                    return 1;
                if (Power < temp.Power)
                    return -1;
                else
                    return 0;
            }
            throw new ArgumentException("Parameter is not a Asteroid!");
        }
    }
}
```

Посмотрим на обобщенный аналог этого же интерфейса:

```
public interface IComparable<T>
{
   int CompareTo(T obj);
}
```

В этом случае код реализации будет значительно яснее:

```
int IComparable<Asteroid>.CompareTo(Asteroid obj)
{
    if (Power > obj.Power)
        return 1;
    if (Power < obj.Power)
        return -1;
    return 0;
}
```

Здесь не нужно проверять, относится ли входной параметр к типу Asteroid, потому что он может быть только им.

Делегаты

Делегат – это вид класса, предназначенный для хранения ссылок на методы. Как и любой другой класс, его можно передать в качестве параметра, а затем вызвать содержащийся в нем метод по ссылке. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка.

Пример описания делегата:

```
public delegate int D(int i,int j);
```

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие **int** и принимающие два параметра типа **int**.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод **Invoke**.

Использование делегатов

Чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

- Чтобы определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- Обеспечивать связь между объектами по типу «источник-наблюдатель»;

- Создавать универсальные методы, в которые можно передавать другие методы;
- Поддерживать механизм обратных вызовов.

Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается функциональная параметризация: в метод можно передавать не только данные, но и функции их обработки.

Простейший пример универсального метода – метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции:

```
using System;
// Пример использования делегата
// Передача делегата через список параметров
// из книги Татьяны Павловской «C#. Программирование на языке высокого уровня» (2009 г.)
namespace DelegatesAndEvents_010
{
    public delegate double Fun(double x);
    class Program
    {
        public static void Table(Fun f, double x, double b)
        {
            Console.WriteLine("----- X ----- Y -----");
            while (x <= b)
            {
                Console.WriteLine("| {0,8:0.000} | {1,8:0.000} |", x, f?.Invoke(x)); // Прежде чем вызвать функцию, обязательно проверяем на существование ссылки на объект
                x += 1;
            }
            Console.WriteLine("----------------------");
        }
        public static double Simple(double x)
        {
            return x * x;
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Таблица функции Sin:");
            Table(Math.Sqrt, -2, 2);
            Console.WriteLine("Таблица функции Simple:");
            Table(Simple, 0, 3);
        }
    }
}
```

Групповая адресация

Делегаты **.NET** обладают встроенной возможностью группового вызова. Другими словами, объект делегата может поддерживать целый список методов для вызова. Для добавления нескольких методов к объекту делегата используется перегруженная операция **+=**, а не прямое присваивание.

```
using System;
namespace ConsoleApplication1
{
    delegate void OpStroke (ref int[] arr);
    public class ArrOperation
    {
        public static void WriteArray(ref int[] arr)
        {
            Console.WriteLine("Исходный массив: ");
            foreach (int i in arr)
                Console.Write("{0}\t", i);
            Console.WriteLine();
        }
        // Сортировка массива
        public static void IncSort(ref int[] arr)
        {
            int j,k;
            for (int i = 0; i < arr.Length - 1; i++)
            {
                j = 0;
                do
                {
                    if (arr[j] > arr[j + 1])
                    {
                         k = arr[j];
                         arr[j] = arr[j+1];
                         arr[j+1] = k;
                     }
                     j++;
                }
                while (j < arr.Length - 1);
            }
            Console.WriteLine("Отсортированный массив в большую сторону: ");
            foreach (int i in arr)
                Console.Write("{0}\t",i);
            Console.WriteLine();
        }
        public static void DecSort(ref int[] arr)
        {
            int j, k;
            for (int i = 0; i < arr.Length - 1; i++)
            {
                j = 0;
                do
                {
                    if (arr[j] < arr[j + 1])
```

```
                {
                    k = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = k;
                }
                j++;
            }
            while (j < arr.Length - 1);
        }
        Console.WriteLine("Отсортированный массив в меньшую сторону: ");
        foreach (int i in arr)
            Console.Write("{0}\t", i);
        Console.WriteLine();
    }
    // Заменяем нечетные числа четными и наоборот
    public static void ChetArr(ref int[] arr)
    {
        Console.WriteLine("Четный массив: ");
        for (int i = 0; i < arr.Length; i++)
            if (arr[i] % 2 != 0)
                arr[i] += 1;
        foreach (int i in arr)
            Console.Write("{0}\t", i);
        Console.WriteLine();
    }
    public static void NeChetArr(ref int[] arr)
    {
        Console.WriteLine("Нечетный массив: ");
        for (int i = 0; i < arr.Length; i++)
            if (arr[i] % 2 == 0)
                arr[i] += 1;

        foreach (int i in arr)
            Console.Write("{0}\t", i);
        Console.WriteLine();
    }
}
class Program
{
    static void Main()
    {
        int[] myArr = new int[6] { 2, -4, 10, 5, -6, 9 };
        // Структурируем делегаты
        OpStroke Del;
        OpStroke Wr = ArrOperation.WriteArray;
        OpStroke OnSortArr = ArrOperation.IncSort;
        OpStroke OffSortArr = ArrOperation.DecSort;
        OpStroke ChArr = ArrOperation.ChetArr;
        OpStroke NeChArr = ArrOperation.NeChetArr;
        // Групповая адресация
        Del = Wr;
        Del += OnSortArr;
        Del += ChArr;
```

```
            Del += OffSortArr;
            Del += NeChArr;
            // Выполняем делегат
            Del?.Invoke(ref myArr);
            Console.ReadLine();
        }
    }
}
```

Удаление целей из списка вызовов делегата

Для удаления метода из списка делегатов можно воспользоваться перегруженной операцией **-=**.

Удаление метода нечетного массива:

```
Del-=NeChArr
```

Обобщенные делегаты

Язык C# позволяет определять обобщенные типы делегатов. Например, необходимо определить делегат, который может вызывать любой метод, возвращающий **void** и принимающий единственный параметр. Если передаваемый аргумент может изменяться – это моделируется через параметр типа. Рассмотрим код нового консольного приложения **GenericDelegate**:

```
using System;
namespace GenericDelegate
{
// Этот обобщенный делегат может вызывать любой метод, который возвращает
void и принимает
//  единственный параметр типа
    public delegate void MyGenericDelegate<T>(T arg);
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Generic Delegates *****\n");
// Зарегистрировать цели
            MyGenericDelegate<string> strTarget = new
MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");
// А можно просто указать метод
            MyGenericDelegate<int> intTarget = IntTarget;
            intTarget(9);
            Console.ReadLine();
        }
        static void StringTarget(string arg)
        {
            Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
        }
        static void IntTarget(int arg)
        {
            Console.WriteLine("++arg is: {0}", ++arg);
        }
    }
}
```

Обобщенные делегаты Action<>, Func<> и Predicate<>

Когда необходимо использовать делегаты для включения обратных вызовов в приложениях, ранее мы выполняли следующие шаги:

- определение специального делегата, соответствующего формату метода, на который он указывает;
- создание экземпляра специального делегата с передачей имени метода в качестве аргумента конструктора;
- косвенное обращение к методу через вызов **Invoke ()** на объекте делегата.

При таком подходе в конечном итоге, как правило, получается несколько специальных делегатов, которые никогда не могут применяться за пределами текущей задачи (например, **MyGenericDelegate<T>**, **CarEngineHandler** и т.д.). Может случаться так, что в проекте требуется специальный, уникально именованный делегат, но в других ситуациях точное имя делегата несущественно. Во многих случаях необходим просто «некоторый делегат», принимающий набор аргументов и, возможно, возвращающий значение, отличное от **void**. В таких ситуациях можно воспользоваться встроенными в платформу делегатами **Action<>** и **Func<>**. Посмотрим на них в действии – создадим проект консольного приложения **ActionAndFuncDelegates**.
Обобщенный делегат **Action<>** определен в пространствах имен **System** внутри сборок **mscorlib.dll** и **System.Core.dll**. Его можно применять для указания на метод, который принимает вплоть до 16 аргументов (этого должно быть достаточно) и возвращает **void**. Поскольку **Action<>** является обобщенным делегатом, понадобится также указывать типы каждого параметра.

Модифицируйте код класса **Program**, определив новый статический метод, который принимает порядка трех уникальных параметров. Например:

```
// Это цель для делегата Action<>.
static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
{
// Установить цвет текста консоли
  ConsoleColor previous = Console.ForegroundColor;
  Console.ForegroundColor = txtColor;
  for (int i = 0; i < printCount; i++)
  {
    Console.WriteLine(msg);
  }
// Восстановить цвет
  Console.ForegroundColor = previous;
}
```

Теперь вместо построения специального делегата вручную для передачи потока программы методу **DisplayMessage()** мы можем использовать готовый делегат **Action<>**, как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Action and Func *****");
  // Использовать делегат Action<> для указания на DisplayMessage.
  Action<string, ConsoleColor, int> actionTarget =
  new Action<string, ConsoleColor, int>(DisplayMessage);
  actionTarget("ActionMessage!", ConsoleColor.Yellow, 5);
  Console.ReadLine();
}
```

Применение **Action<>** не заставляет беспокоиться об определении специального делегата. Но он может указывать только на методы, которые имеют возвращаемое значение **void**. Если нужно указывать на метод с другим возвращаемым значением (и нет желания заниматься написанием собственного делегата), можно прибегнуть к делегату **Func<>**.

Обобщенный делегат **Func<>** может указывать на методы, которые (подобно **Action<>**) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение.

Добавим следующий новый метод в класс **Program**:

```
// Цель для делегата Func<>.
static int Add(int x, int y) => x + y;
```

Учтите, что финальный параметр типа **Func<>** – это всегда возвращаемое значение метода. Чтобы закрепить этот момент, предположим, что в классе **Program** также определен следующий метод:

```
static string SumToString(int x, int y) => (x + y).ToString();
```

Теперь метод **Main()** может вызывать каждый из этих методов, как показано ниже:

```
Func<int, int, int> funcTarget = Add;
int result = funcTarget.Invoke(40, 40);
Console.WriteLine($"40 + 40 = {result}");
Func<int, int, string> funcTarget2 = SumToString;
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

Делегаты **Action<>** и **Func<>** могут устранить шаг по ручному определению специального делегата. Учитывая это, следует ли ими пользоваться всегда? Это зависит от ситуации. Во многих случаях **Action<>** и **Func<>** будут предпочтительным вариантом. Тем не менее, если нужен делегат со специфическим именем, которое, как вы чувствуете, помогает лучше отразить предметную область, то построение специального делегата сведется к одиночному оператору кода.

Делегат **Predicate<T>**, как правило, используется для сравнения, сопоставления некоторого объекта **T** определенному условию. В качестве выходного результата возвращается значение **true**, если условие соблюдено, и **false**, если не соблюдено:

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };
Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

Паттерн «наблюдатель»

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект-источник при изменении своего состояния, которое значимо для других объектов, посылает им уведомления. Эти объекты называются наблюдателями. Получив уведомления, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние.

Наблюдатель (observer) определяет между объектами зависимость типа «один ко многим», так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются.

Рассмотрим пример:

```
using System;
namespace Delegates_Observer
{
    public delegate void MyDelegate(object o);
    class Source
    {
        MyDelegate functions;

        public void Add(MyDelegate f)
        {
            functions += f;
        }

        public void Remove(MyDelegate f)
        {
            functions -= f;
        }

        public void Run()
        {
            Console.WriteLine("RUNS!");
            if (functions != null) functions(this);
        }
    }
    class Observer1 // Наблюдатель 1
    {
        public void Do(object o)
        {
            Console.WriteLine("Первый. Принял, что объект {0} побежал", o);
        }
    }
    class Observer2 // Наблюдатель 2
    {
        public void Do(object o)
        {
            Console.WriteLine("Второй. Принял, что объект {0} побежал", o);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Source s = new Source();
            Observer1 o1 = new Observer1();
            Observer2 o2 = new Observer2();
            MyDelegate d1=new MyDelegate(o1.Do);
            s.Add(d1);
            s.Add(o2.Do);
            s.Run();
            s.Remove(o1.Do);
            s.Run();
        }
    }
}
```

События

Рассмотрим подробнее события среды **.Net Framework** в классе **Timer**.

Событие – это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, которые являются наблюдателями события, активизируются методы-обработчики этого события.

События построены на основе делегатов: с их помощью вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание методов, инициирующих событие.

Пример описания делегата и соответствующего ему события:

```
public delegate void Delegat();
class A
{
  public event Delegate Event;
}
```

Событие – это удобная абстракция для программиста. На самом деле, событие состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.

Давайте разберем еще один пример.

Пример Lesson3

```
using System;
namespace Delegates_Observer
{
    public delegate void MyDelegate(object o);
    class Source
    {
       public event MyDelegate Run;

        public void Start()
        {
            Console.WriteLine("RUN");
            if (Run != null) Run(this);
        }
    }
    class Observer1 // Наблюдатель 1
    {
        public void Do(object o)
        {
            Console.WriteLine("Первый. Принял, что объект {0} побежал", o);
        }
    }
    class Observer2 // Наблюдатель 2
    {
        public void Do(object o)
        {
            Console.WriteLine("Второй. Принял, что объект {0} побежал", o);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Source s = new Source();
            Observer1 o1 = new Observer1();
            Observer2 o2 = new Observer2();
            MyDelegate d1 = new MyDelegate(o1.Do);
            s.Run += d1;
            s.Run += o2.Do;
            s.Start();
            s.Run -= d1;
            s.Start();
        }
    }
}
```

Пример создания событий

```
using System.Collections.Generic;
using System.Windows.Forms;

namespace GeekBrains
{
    class Source
    {
        public delegate void Message(string message);
        private event Message _message;

        private readonly List<string> _user = new List<string>();

        public Source()
        {
            _user.AddRange(new[] { "Ivan", "Roman", "Stepan" });
        }

        public void RemoveUser(string nameUser, Message message)
        {
            _message = message;
            if (_user.Contains(nameUser))
            {
                _user.Remove(nameUser);
                _message?.Invoke($"Пользователь {nameUser} удален");
            }
            else
            {
                _message?.Invoke($"Пользователь {nameUser} не найден");
            }
        }
    }
    class Program
    {
        static void Main()
        {
            Source source = new Source();
            source.RemoveUser("Ivan", Message);
        }

        private static void Message(string message)
        {
            MessageBox.Show(message);
        }
    }
}
```

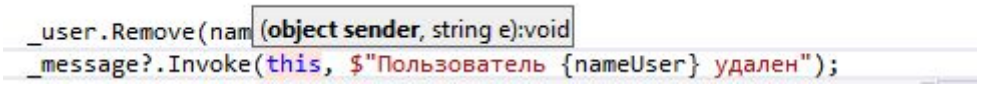
Данный пример демонстрирует передачу данных между классами. Но, как правило, нам нужно знать, от кого пришло данное событие.

```
using System.Collections.Generic;
using System.Windows.Forms;

namespace GeekBrains
{
    class Source
    {
// Добавили новый параметр
        public delegate void Message(object obj, string message);
        private event Message _message;

        private List<string> _user = new List<string>();

        public Source()
        {
            _user.AddRange(new[] { "Ivan", "Roman", "Stepan" });
        }

        public void RemoveUser(string nameUser, Message message)
        {
            _message = message;
            if (_user.Contains(nameUser))
            {
                _user.Remove(nameUser);
// Изменился вызов события
                _message?.Invoke(this, $"Пользователь {nameUser} удален");
            }
            else
            {
                _message?.Invoke(this, $"Пользователь {nameUser} не найден");
// Изменился вызов события
            }
        }
    }
    class Program
    {
        static void Main()
        {
            Source source = new Source();
            source.RemoveUser("Ivan", Message);
        }

        private static void Message(object o, string message)
        {
            MessageBox.Show(message);
        }
    }
}
```

Делегат **EventHandler**:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace GeekBrains
{
    class Source
    {
        private event EventHandler<string> _message; // Объявляем Делегат
EventHandler

        private List<string> _user = new List<string>();

        public Source()
        {
            _user.AddRange(new[] { "Ivan", "Roman", "Stepan" });
        }

        public void RemoveUser(string nameUser, EventHandler<string> message)
        {
            _message = message;
            if (_user.Contains(nameUser))
            {
                _user.Remove(nameUser);
                _message?.Invoke(this, $"Пользователь {nameUser} удален");
            }
            else
            {
                _message?.Invoke(this, $"Пользователь {nameUser} не найден");
            }
        }
    }
    class Program
    {
        static void Main()
        {
            Source source = new Source();
            source.RemoveUser("Ivan", Message);
        }

        private static void Message(object o, string message)
        {
            MessageBox.Show(message);
        }
    }
}
```

Мы избавились от делегата, но потеряли читабельность.



Неочевидно, какие параметры нужно передавать. Поэтому мы введем дополнительный класс:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace GeekBrains
{
    public class RemoveUserEventArgs : EventArgs
    {
        public string Message { get; }
        // Можем дописать сколько угодно свойств
        public RemoveUserEventArgs(string message)
        {
            Message = message;
        }
    }

    class Source
    {
        private event EventHandler<RemoveUserEventArgs> _message;

        private List<string> _user = new List<string>();

        public Source()
        {
            _user.AddRange(new[] { "Ivan", "Roman", "Stepan" });
        }

        public void RemoveUser(string nameUser,
EventHandler<RemoveUserEventArgs> message)
        {
            _message = message;
            if (_user.Contains(nameUser))
            {
                _user.Remove(nameUser);
                _message?.Invoke(this, new RemoveUserEventArgs($"Пользователь
{nameUser} удален"));
            }
            else
            {
                _message?.Invoke(this, new RemoveUserEventArgs($"Пользователь
{nameUser} не найден"));
            }
        }
    }
    class Program
```

```
    {
        static void Main()
        {
            Source source = new Source();
            source.RemoveUser("Ivan", Message);
        }

        // Привели к виду обработчиков событий Windows Forms или WPF
        private static void Message(object o, RemoveUserEventArgs message)
        {
            MessageBox.Show(message.Message);
        }
    }
}
```

Анонимные методы

С делегатами тесно связано понятие анонимных методов. Анонимные методы представляют сокращенную запись методов. Иногда они нужны для обработки одного события и больше нигде не используются. Анонимные методы позволяют встроить код там, где он вызывается. Практически, это тот же самый метод, только встроенный в код. Встраивание происходит с помощью ключевого слова **delegate**, после которого идет список параметров и далее сам код анонимного метода. В отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки. Если для анонимного метода не требуется параметров, он используется без скобок.

```
using System.Windows.Forms;

class Program
{
    static void Main()
    {
        Source source = new Source();
        source.RemoveUser("Ivan", delegate (object sender, RemoveUserEventArgs
args) { MessageBox.Show(args.Message); });
    }
}
```

Лямбды

Лямбда-выражения представляют упрощенную запись анонимных методов. Позволяют создать емкие лаконичные методы, которые могут возвращать значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора **=>** определяется список параметров, а справа – блок выражений, использующий эти параметры: **(список_параметров) => выражение**. Как и делегаты, лямбда-выражения можно передавать в качестве параметров методу:

```
using System.Windows.Forms;

class Program
{
    static void Main()
    {
        Source source = new Source();
        source.RemoveUser("Ivan", (sender, args) =>
        {
            MessageBox.Show(args.Message);
        });
    }
}
```

Примеры

Постройте таблицу значений функции **y=f(x)** для **x[a, b]** с шагом **h**. Если в некоторой точке **x** функция не определена, то выведите на экран сообщение об этом.

Замечание. *При решении данной задачи использовать вспомогательный метод f(x), реализующий заданную функцию, а также проводить обработку возможных исключений.*

$$y = \frac{1}{(1+x)^2}$$

```
using System;
namespace Hello
{
	class Program
	{
	static double f(double x)
	{
		try
		{
 // Если x не попадает в область определения, то генерируется исключение
		 if (x == -1) throw new Exception();
		else return 1 / Math.Pow(1 + x, 2);
		}
		catch
		{
		throw;
		 }
	}
	static void Main(string[] args)
	{
		try
		{
		Console.Write("a=");
		double a = double.Parse(Console.ReadLine());
		Console.Write("b=");
		double b = double.Parse(Console.ReadLine());
		Console.Write("h=");
		double h = double.Parse(Console.ReadLine());
		for (double i = a; i <= b; i += h)
			try
			{
				Console.WriteLine("y({0})={1:f4}", i, f(i));
			}
			catch
			{
				Console.WriteLine("y({0})=error", i);
			}
		}
		catch (FormatException)
		{
		Console.WriteLine("Неверный формат ввода данных");
		}
		catch
		{
		    Console.WriteLine("Неизвестная ошибка");
		}
	 }
   }
}
```

Использование делегата .Net для конвертации массива

```
using System;
/* Условие задачи: напишите программу поиска номера первого из двух
последовательных элементов в целочисленном массиве из 30 элементов, сумма
которых максимальна. Если таких элементов несколько, следует вывести номер
первой пары. */
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] s = Console.ReadLine().Split(new char[] {' '},
StringSplitOptions.RemoveEmptyEntries);
            int[] a = Array.ConvertAll(s, new
Converter<string,int>(Convert.ToInt32));
            int imax = 0;
            for (int i = 1; i < a.Length - 1; i++)
                if (a[imax] + a[imax + 1] < a[i] + a[i + 1]) imax = i;
            Console.Write(imax + 1);
        }
    }
}
```

Сортировка с использованием делегата

Пример решения задачи ЕГЭ С4 на C# с использованием делегата:

```
using System;
using System.Collections.Generic;
using System.IO;
namespace HW_TaskEGE
{
    class Element
    {
        private string _fio;
        private int _ball;
        public Element(string fio, int ball)
        {
            _fio = fio;
            _ball = ball;
        }
        public string FIO => _fio;
        public int Ball => _ball;
    }
    class Program
    {
        static int MyDelegat(Element el1, Element el2)
```

```
        {
            if (el1.Ball > el2.Ball) return 1;
            if (el1.Ball < el2.Ball) return -1;
            return 0;
        }
        static void Main(string[] args)
        {
            List<Element> list = new List<Element>();
            using (StreamReader sr = new StreamReader("data.txt"))
            {
                int n = int.Parse(sr.ReadLine() ?? throw new
InvalidOperationException());
                for (int i = 0; i < n; i++)
                {
                    string[] s = sr.ReadLine()?.Split(' ');
                    int ball = int.Parse(s[2]) + int.Parse(s[3]) +
int.Parse(s[4]);
                    list.Add(new Element(s[0] + " " + s[1], ball));
                }
            }
            list.Sort(MyDelegat);
            foreach (var v in list)
            {
                Console.WriteLine($"{v.FIO,20}{v.Ball,10}");
            }
            Console.WriteLine();
            int ball2 = list[2].Ball;
            foreach (var v in list)
            {
                if (v.Ball <= ball2)
Console.WriteLine($"{v.FIO,20}{v.Ball,10}");
            }
        }
    }
}
```

Практика

Попрактикуемся в использовании событий для создания управляемого корабля. В **.Net Framework** довольно много уже встроенных событий для обработки ситуаций, возникающих во время выполнения программы. Для управления кораблем мы используем события **KeyDown**.

Сначала добавим класс **Ship**, который будет представлять наш космический корабль:

```
using System.Drawing;

namespace MyGame
{
    class Ship : BaseObject
    {
        private int _energy = 100;
        public int Energy => _energy;

        public void EnergyLow(int n)
        {
            _energy -= n;
        }
        public Ship(Point pos, Point dir, Size size) : base(pos, dir, size)
        {
        }
        public override void Draw()
        {
            Game.Buffer.Graphics.FillEllipse(Brushes.Wheat, Pos.X, Pos.Y,
Size.Width, Size.Height);
        }
        public override void Update()
        {
        }
        public void Up()
        {
            if (Pos.Y > 0) Pos.Y = Pos.Y - Dir.Y;
        }
        public void Down()
        {
            if (Pos.Y < Game.Height) Pos.Y = Pos.Y + Dir.Y;
        }
        public void Die()
        {
        }
    }

}
```

Создадим в классе **Game** статический объект **ship**:

```
private static Ship _ship = new Ship(new Point(10, 400), new Point(5, 5), new
Size(10, 10));
```

Далее в методе **Init** класса **Game** добавим обработчики событий на **KeyDown**:

```
form.KeyDown += Form_KeyDown;
```

И сам метод **Form_KeyDown**:

```
private static void Form_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.ControlKey) _bullet = new Bullet(new
Point(_ship.Rect.X + 10, _ship.Rect.Y + 4), new Point(4, 0), new Size(4, 1));
    if (e.KeyCode == Keys.Up) _ship.Up();
    if (e.KeyCode == Keys.Down) _ship.Down();
}
```

Методы **Draw** и **Update** придется существенно переработать, чтобы учитывать столкновения.

Добавим вывод энергии корабля в метод **Draw** класса **Game**:

```
public static void Draw()
{
    Buffer.Graphics.Clear(Color.Black);
    foreach (BaseObject obj in _objs)
        obj.Draw();
    foreach (Asteroid a in _asteroids)
    {
        a?.Draw();
    }
    _bullet?.Draw();
    _ship?.Draw();
    if (_ship != null)
        Buffer.Graphics.DrawString("Energy:" + _ship.Energy,
SystemFonts.DefaultFont, Brushes.White, 0, 0);
    Buffer.Render();
}

public static void Update()
{
    foreach (BaseObject obj in _objs) obj.Update();
    _bullet?.Update();
    for (var i = 0; i < _asteroids.Length; i++)
    {
        if (_asteroids[i] == null) continue;
        _asteroids[i].Update();
        if (_bullet != null && _bullet.Collision(_asteroids[i]))
        {
            System.Media.SystemSounds.Hand.Play();
            _asteroids[i] = null;
            _bullet = null;
            continue;
        }
        if (!_ship.Collision(_asteroids[i])) continue;
        var rnd = new Random();
        _ship?.EnergyLow(rnd.Next(1, 10));
        System.Media.SystemSounds.Asterisk.Play();
        if (_ship.Energy <= 0) _ship?.Die();
    }
}
```

Добавим обработчик событий гибель корабля. Для этого в файл **BaseObject.cs** добавим делегат:

```
public delegate void Message();
```

Внутри класса **Ship** создадим статическое событие:

```
public static event  Message MessageDie;
```

Когда корабль погибает, вызываем это событие:

```
public void Die()
{
    MessageDie?.Invoke();
}
```

Создадим в классе **Game** метод **Finish**. Чтобы он заработал, **Timer** нужно вынести из метода **Init** в класс **Game**:

```
static class Game
{
   private static Timer _timer = new Timer();
   public static Random Rnd = new Random();
...
}
```

```
public static void Finish()
{
    _timer.Stop();
    Buffer.Graphics.DrawString("The End", new Font(FontFamily.GenericSansSerif,
60, FontStyle.Underline), Brushes.White, 200, 100);
    Buffer.Render();
}
```

В методе **Init** класса **Game** подпишемся на это событие:

```
Ship.MessageDie += Finish;
```

Статический импорт

С версии 6.0 в язык C# была добавлена возможность импорта функциональности классов:

```
using System;
using static System.Math;
using static System.Console;

namespace GeekBrains
{
    class Program
    {
        static void Main()
        {
            var x = Int32.Parse(ReadLine()); // Вместо Console.ReadLine()
            var res = 1 / Pow(1 + x, 2);     // Вместо Math.Pow()
            WriteLine(res);                  // Вместо Console.WriteLine()
        }
    }
}
```

Практическое задание

1. Добавить космический корабль, как описано в уроке.
2. Доработать игру «Астероиды»:
    a. Добавить ведение журнала в консоль с помощью делегатов;
    b. * добавить это и в файл.
3. Разработать аптечки, которые добавляют энергию.
4. Добавить подсчет очков за сбитые астероиды.
5. * Добавить в пример **Lesson3** обобщенный делегат.

Дополнительные материалы

1. yield (справочник по C#)
2. https://msdn.microsoft.com/ru-ru/library/dd799517(v=vs.110).aspx

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Татьяна Павловская. Программирование на языке высокого уровня. – 2009 г.
2. Эндрю Троелсен. Язык программирования C# 5.0 и платформа .NET 4.5. –2013 г.

3. Герберт Шилдт. C# 4.0. Полное руководство.
4. MSDN.