Урок 3

Объектно - Ориентированное Программирование

Обобщения. Делегаты и события. Паттерн “наблюдатель”.

Обобщения

Обобщённое программирование (англ. generic programming) — парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.
Продемонстрируем пример не обобщенной реализации функции обмена значениями двух переменных Swap с использованием перегруженных методов. В C# за счет использования перегруженных методов, мы можем создавать функции с одинаковыми именами, но с разными параметрами функций, что сильно упрощает жизнь.

```
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;
}
```

А вот пример реализации Swap при помощи обобщения.

```
static void Swap<T>(ref  T A, ref  T B)
{
    T t;
    t = A;
    A = B;
    B = t;
}
```

Видно, что обобщения ещё более упрощают работу программиста.

После появления первого выпуска платформы .NET программисты часто использовали пространство имён System.Collections для получения более гибкого способа управления данными в приложениях. Однако, начиная с версии .NET 2.0, язык программирования C# был расширен поддержкой средства, которое называется обобщением (generic). Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имён, связанным с коллекциями — System.Collections.Generic.

Термин обобщение, по существу, 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, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

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

Но дело в том, что в таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа object в конкретный тип данных требовалось приведение типов. А это служило потенциальным источником ошибок из-за того, что приведение типов могло быть неумышленно выполнено неверно. Это затруднение позволяют преодолеть обобщения, обеспечивая типовую безопасность, которой раньше так недоставало. Кроме того, обобщения упрощают весь процесс, поскольку исключают 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.

Рассмотрим не обобщенную реализацию интерфейса IComparable. Для этого в класс Asteroid добавим поле Power - сколько нужно выстрелов для его разрушения

```
// Создаем класс Asteroid, так как мы теперь не можем создавать объекты абстрактного класса BaseObject
    class Asteroid : BaseObject, ICollision, IComparable
    {
        public int Power { get; set; } = 3;
        public Asteroid(Point pos, Point dir, Size size) : base(pos, dir, size)
        {
            Power = Game.rnd.Next(1, 4);// От 1 до 3 выстрелов для разрушения
        }
        public override void Draw()
        {
            Game.buffer.Graphics.DrawEllipse(Pens.White, pos.X, pos.Y, size.Width, size.Height);
        }
        int IComparable.CompareTo(object obj)
        {
            Asteroid temp = obj as Asteroid;
            if (temp != null)
            {
                if (this.Power > temp.Power)
                    return 1;
                if (this.Power < temp.Power)
                    return -1;
                else
                    return 0;
            }
            else
                throw new ArgumentException("Parameter is not a Asteroid!");
        }
```

Теперь посмотрим на обобщённый 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 (this.Power > obj.Power)
        return 1;
     if (this.Power < obj.Power)
        return -1;
     else
        return 0;
  }
```

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

Делегаты

Делегат - это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать содержащийся в нем метод. Делегаты используются для поддержки событий, а так же 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 имён или внутри класса.

Использование делегатов

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

Делегаты применяются в основном для следующих целей:

- Получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- Обеспечения связи между объектами по типу “источник-наблюдатель”;

- Создания универсальных методов, в которые можно передавать другие методы;
- Поддержки механизма 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 различные данные, но и различные функции их обработки.

В качестве простейшего примера универсального метода можно привести 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
// C# Программирование на языке высокого уровня Т.А. Павловская  Питер 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(x));
            x += 1;
         }
         Console.WriteLine("----------------------");
      }
      public static double Simple(double x)
      {
         return x * x;
      }
      static void Main(string[] args)
      {
         Console.WriteLine("Таблица функции Sin:");
         Table(new Fun(Math.Sqrt), -2, 2);
         Console.WriteLine("Таблица функции Simple:");
         Table(new Fun(Simple), 0, 3);
      }
   }
}
```

Групповая адресация

Делегаты .NET обладают встроенной возможностью группового вызова. Другими словами, объект делегата может поддерживать целый список методов для вызова, а не просто единственный метод. 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 нечётные числа чётными 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(ref myArr);
            Console.ReadLine();
        }
    }
}
```

Удаление целей из списка вызовов делегата

Для удаления метода из списка делегатов можно воспользоваться перегруженную операцию -=

Например, удаление метода Нечетный массив

```
Del-=NeChArr
```

Обобщенные делегаты

Язык C# позволяет определять обобщённые типы делегатов. Например, предположим, что необходимо определить делегат, который может вызывать любой метод, возвращающий void и принимающий единственный параметр. Если передаваемый аргумент может изменяться, это моделируется через параметр типа. Для иллюстрации рассмотрим следующий код нового консольного приложения по имени GenericDelegate:

```
using System;
namespace GenericDelegate
{
 // Этот обобщённый 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");
         MyGenericDelegate<int> intTarget = new MyGenericDelegate<int>(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<> и Func<>

Ранее вы видели, что когда необходимо использовать делегаты для включения обратных вызовов в приложениях, обычно выполнялись 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.

В случае принятия такого подхода, как правило, в конечном итоге получается несколько специальных делегатов, которые никогда не могут применяться за пределами текущей задачи (например,

MyGenericDelegate<T>, CarEngineHandler и т.д.). Хотя может случаться так, что в проекте требуется специальный, уникально именованный делегат, в других ситуациях точное имя делегата является несущественным. 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<>. Для иллюстрации их удобства создадим проект консольного приложения по имени ActionAndFuncDelegates.
Обобщенный делегат Action<> определён в пространствах имён System внутри сборок mscorlib.dll и System.Core.dll. Этот обобщенный делегат можно применять для “указания на” метод, который принимает вплоть до 16 аргументов (чего должно быть достаточно!) и возвращает void. Вспомните, что поскольку Action<> является обобщённым делегатом, понадобится также указывать типы каждого параметра.
Модифицируйте код класса Program, определив новый статический метод, который принимает три (или около того) уникальных параметра, 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();
}
```

Как видите, применение делегата Action<> не заставляет беспокоиться об определении специального делегата. Однако вспомните, что делегат Action<> 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<>.

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

В целях иллюстрации добавим следующий новый метод в класс Program:

```
// Цель для делегата Func<>.
static int Add(int x, int y)
{
  return 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)
{
  return (x + y).ToString();
}
```

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

```
Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);
Func<int, int, string> funcTarget2 = new Func<int, int, string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

Таким образом, учитывая, что делегаты Action<> и Func<> могут устранить шаг по ручному определению специального делегата, вас может интересовать, следует ли ими пользоваться всегда. Ответом будет, как и в случае многих аспектов программирования — в зависимости 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.

Паттерн “наблюдатель”

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый источником, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются наблюдателями. Получив уведомления, наблюдатель опрашивает источник, чтобы синхронизировать с ним своё 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 Register(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();
         s.Register(new MyDelegate(o1.Do));
         s.Register(new MyDelegate(o2.Do));
         s.Run();
      }
   }
}
```

События

Мы с вами уже использовали события .среды Net Framework в класса Timer. Теперь давайте познакомимся с ними поближе.

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 помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание метода (методов), инициирующих событие

Пример описания делегата и соответствующего ему события:

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

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

Давайте разберем еще один пример.

Пример Lesson3

```
using System;
// Демонстрация описания собственного события
namespace Lesson3
{
    class ClassCounter  //Это класс - в котором происходит событие.
    {
 // Синтаксис по сигнатуре метода, на который мы создаём делегат:
 // delegate <выходной тип> ИмяДелегата(<тип входных параметров>);
// Мы создаём на void Message(). Он должен запуститься, когда условие выполнится.
        public delegate void MethodContainer();
// Событие OnCount с типом делегата MethodContainer.
        public event MethodContainer onCount;
        public void Count()
        {
 // Здесь будет производиться счёт
            for (int i = 0; i < 20; i++)
            {
                if (i == 10)
                {
                    onCount();
                    Console.Write(i + " ");
                }
            }
        }
    }
// Это класс, реагирующий на событие (счет равен 71) записью строки в консоли.
    class Handler_I
    {
        public void Message()
        {
 // Не забудьте using System для вывода в консольном приложении
            Console.WriteLine("Я знаю, уже 10!");
        }
    }
    class Handler_II
    {
        public void Message()
        {
            Console.WriteLine("И я знаю, что уже 10!");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            ClassCounter Counter = new ClassCounter();
            Handler_I handler_I = new Handler_I();
            Handler_II handler_II = new Handler_II();
 // Подписались на событие
            Counter.onCount += handler_I.Message;
            Counter.onCount += handler_II.Message;
            Counter.Count();
        }
    }
}
```

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, то 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
    {
        string fio;
        int ball;
        public Element(string fio, int ball)
        {
            this.fio = fio;
            this.ball = ball;
        }
        public string FIO
        {
            get { return fio; }
        }
        public int Ball
        {
            get { return 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>();
            StreamReader sr = new StreamReader("data.txt");
            int N = int.Parse(sr.ReadLine());
            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));
            }
            sr.Close();
            list.Sort(new Comparison<Element>(MyDelegat));
            foreach(var v in list)
            {
                Console.WriteLine("{0,20}{1,10}",v.FIO,v.Ball);
            }
            Console.WriteLine();
            int ball2 = list[2].Ball;
            foreach (var v in list)
            {
                if (v.Ball<=ball2) Console.WriteLine("{0,20}{1,10}", v.FIO, v.Ball);
            }
        }
    }
}
```

Практика

Попрактикуемся в использовании событий для создания управляемого корабля. В .Net Framework довольно много уже встроенных событий для обработки различных событий возникающий во время выполнения программы. Для управления кораблем мы используем события KeyDown. Для начала добавим класс Ship, который будет представлять наш космический корабль

```
class Ship: BaseObject
  {
     int energy=100;
     public int Energy
     {
        get { return 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.

```
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 событие KeyDown

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

И сам метод Form_KeyDown

```
static private 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 придётся существенно переработать, чтобы учитывать столкновения.

И добавим вывод энергии корабля в метод Draw класса Game

```
static public void Draw()
 {
     buffer.Graphics.Clear(Color.Black);
     foreach (BaseObject obj in objs)
         obj.Draw();
     foreach (Asteroid a in asteroids)
          if (a!=null) a.Draw();
     if (bullet!=null) bullet.Draw();
     ship.Draw();
     buffer.Graphics.DrawString("Energy:" + ship.Energy, SystemFonts.DefaultFont, Brushes.White, 0, 0);
     buffer.Render();
 }
static public void Update()
{
     foreach (BaseObject obj in objs) obj.Update();
     if (bullet!=null) bullet.Update();
     for(int i=0;i<asteroids.Length;i++)
     {
         if (asteroids[i] != null)
         {
             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]))
             {
                 ship.EnergyLow(rnd.Next(1, 10));
                 System.Media.SystemSounds.Asterisk.Play();
                 if (ship.Energy <= 0) ship.Die();
             }
         }
     }
 }
```

Добавим обработчик событий, когда корабль погибает.

Для этого в файл BaseObject.cs добавим делегат:

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

Внутри класса Ship создадим статическое событие:

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

Когда корабль погибает вызываем это событие:

```
if (MessageDie != null) MessageDie();
```

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

```
static class Game
 {
       static Timer timer = new Timer();
       public static Random rnd = new Random();
...
}
```

```
static public 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;
```

Домашнее задание

1. а) Добавить в игру “Астероиды” ведение журнала в консоль
   б)*и в файл.
2. Добавьте аптечки, которые добавляют энергии.
3. Добавить подсчет очков за сбитые астероиды.
4. *Добавьте в пример Lesson3 обобщенный делегат.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Т.А. Павловская. “Программирование на языке высокого уровня”, 2009 г.
2. “Язык программирования C# 5.0 и платформа .NET 4.5”. Эндрю Троелсен, Питер 2013 г.
3. Г.Шилдт. “C# 4.0. Полное руководство”.
4. MSDN.