Урок 2

Объектно-ориентированное программирование. Часть 2

Абстрактные методы и классы. Интерфейсы. Стандартные интерфейсы. Исключения.

Абстрактный метод

Абстрактный метод не реализуется в базовом классе, он должен быть переопределен в наследуемом. Абстрактными могут быть также индексаторы и свойства.

```
class abstract  MyClass
    {
        protected  abstract void Show();
    }
```

Абстрактные методы не могут быть приватными (должны быть либо **public**, либо **protected**). По сути, абстрактный метод – это виртуальный метод, только без определения поведения. Подразумевается, что поведение абстрактного метода будет реализовано в классах-наследниках. Абстрактный метод может быть описан только в абстрактном классе.

Абстрактный класс

Программист создает абстрактный класс, чтобы заложить в него логику, которая будет заимствована классами-потомками. Они не могут использоваться для создания объектов, потому что абстрактные классы не завершены – производные классы должны предоставить «недостающие части».

```
namespace Abstract
{
            // Создаем абстрактный класс
   class abstract BaseObject
    {
    }
    class Program
    {
        void static  Main(string[] args)
        {
            // Мы не можем создавать экземпляры абстрактного класса
            // BaseObject obj = new BaseObject();
        }
    }
}
```

Зачем определять класс, экземпляр которого нельзя создать непосредственно? Базовые классы (абстрактные или нет) очень полезны: они содержат общие данные и общую функциональность для унаследованных типов. Используя эту форму абстракции, можно также моделировать общую «идею», а не обязательно конкретную сущность. И хотя непосредственно создать экземпляр абстрактного класса нельзя, он все же присутствует в памяти, когда создан экземпляр его производного класса. Таким образом, совершенно нормально (и принято) для абстрактных классов определять любое количество конструкторов, вызываемых опосредованно при размещении в памяти экземпляров производных классов.

«Астероиды» с использованием абстрактного класса

Что собой представляет базовый объект? Это общая сущность для описания конкретных объектов, поэтому логично сделать его абстрактным:

```
class abstract  BaseObject
```

Добавим слово **abstract** перед названием метода **Draw**. **Virtual** нужно убрать, так как абстрактный метод подразумевает виртуальность. Удалите тело метода **Draw**. Модификатор доступа конструктора абстрактного класса логичнее сделать **protected**, так как создать экземпляр такого класса нельзя, а унаследовать конструктор можно.

```
abstract  class BaseObject
  {
      protected Point Pos;
      protected Point Dir;
      protected Size Size;

      protected BaseObject(Point pos, Point dir, Size size)
      {
          Pos = pos;
          Dir = dir;
          Size = size;
      }
      public abstract  void Draw();
      public virtual  void Update()
      {
          Pos .X = Pos .X + Dir .X;
          if (Pos .X < 0) Pos .X = Game.Width + Size.Width;
      }
  }
```

Теперь у нас нет возможности создавать объекты абстрактного класса **BaseObject**. Абстрактные классы для того и создаются, чтобы в производных классах дописывалась их функциональность. Такая техника дает возможность задавать модель поведения заранее. Само же поведение должно быть реализовано в классах-наследниках. Особенно это прослеживается в абстрактном методе **Draw**, тело которого мы даже не описали (оно и не может быть описано). Абстрактный метод должен быть описан в классах-наследниках. Абстрактными могут быть также свойства, события и индексаторы.

Создадим еще один класс **Asteroid**, который будет реализовывать метод **Draw**:

```
// Создаем класс Asteroid, так как мы теперь не можем создавать объекты
абстрактного класса BaseObject
class Asteroid: BaseObject
{
  public int Power {get;set;}
  public Asteroid(Point pos, Point dir, Size size) : base(pos, dir, size)
  {
     Power=1;
  }
  public override void Draw()
  {
      Game.Buffer.Graphics.FillEllipse(Brushes.White, Pos.X,Pos.Y,
Size.Width,Size.Height);
  }
}
```

Метод **Update** мы можем переопределить или воспользоваться реализацией базового класса.

Также добавим класс **Bullet**, который будет описывать поведение снарядов.

```
class Bullet : BaseObject
{
  public Bullet(Point pos, Point dir, Size size) : base(pos, dir, size)
  {
  }
 public override void Draw()
 {
  Game.Buffer.Graphics.DrawRectangle(Pens.OrangeRed, Pos.X, Pos.Y, Size.Width,
Size.Height);
 }
 public override void Update()
 {
     Pos.X = Pos.X + 3;
  }
}
```

В классе **Game** переделаем метод **Load**, который теперь будет создавать объекты классов **Star**, **Asteroid** и **Bullet**.

Объекты **bullet** и **asteroid** должны быть описаны в классе **Game**.

```
private static Bullet _bullet;
private static Asteroid[] _asteroids;
public static void Load()
{
    _objs = new BaseObject[30];
    _bullet = new Bullet(new Point(0, 200), new Point(5, 0), new Size(4, 1));
    _asteroids = new Asteroid[3];
    var rnd  = new Random();
    for (var i = 0; i < _objs.Length; i++)
    {
        int r = rnd.Next(5, 50);
        _objs[i] = new Star(new Point(1000, rnd.Next(0, Game.Height)), new
Point(-r, r), new Size(3, 3));
    }
    for (var i = 0; i < _asteroids.Length; i++)
    {
        int r = rnd.Next(5, 50);
        _asteroids[i] = new Asteroid(new Point(1000, rnd.Next(0, Game.Height)),
new Point(-r / 5, r), new Size(r, r));
    }
}
```

Интерфейс

Интерфейс похож на класс, но он содержит спецификацию, а не реализацию своих членов. Особенности интерфейсов:

- Члены интерфейса всегда неявно абстрактные. Класс, напротив, может содержать как абстрактные, так и конкретные методы с реализацией;
- Класс (или структура) может реализовать несколько интерфейсов. Класс может быть наследником только одного класса, а структура не допускает наследования вообще (кроме класса **System.ValueType**).

Объявление интерфейса напоминает объявление класса, но не содержит реализации своих членов, поскольку все его члены неявно абстрактные. Эти члены можно реализовать в классах и структурах, реализующих интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, которые не случайно являются членами класса (он может быть абстрактным). Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I** (это не обязательное требование, а стиль программирования). Все члены интерфейса (методы и свойства) не имеют модификаторов доступа. Но фактически, по умолчанию, доступ **public**, так как цель интерфейса – определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Рассмотрим упрощенную версию интерфейса **IEnumerator**, определенную в классе **System. Collections**:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
}
```

Оператор **foreach** применяется только к классам, в которых реализован интерфейс **IENumerator**. Этот пример демонстрирует важность интерфейсов. Они задают только то, что должен выполнять класс, но не говорят, как это делается. Оператор **foreach** может проходить по элементам класса, потому что интерфейс **IEnumerator** обязывает реализовать в классе переход к следующему элементу и получение значения текущего элемента. Все классы, с которыми вы ранее использовали оператор **foreach**, являются наследниками интерфейса **IEnumerator**.

Члены интерфейса неявно всегда открытые и не могут объявлять модификатор доступа. Реализация интерфейса означает открытую реализацию всех его членов.

```
class Countdown : IEnumerator
{
    private int _count = 11;
    public bool MoveNext() => _count-- > 0;// Здесь count-- и сравнение
получившегося значения с 0
    public void Reset() => _count = 0;

    public object Current => _count;
}
```

Объект можно неявно привести к любому интерфейсу, который он реализует.

```
IEnumerator e = new Countdown();
while (e.MoveNext())
Console.Write (e.Current); // 109876543210
```

Полный текст программы:

```
using System;
namespace Interface_sample
{
    public interface IEnumerator
    {
        bool MoveNext();
        object Current { get; }
    }
    internal class Countdown : IEnumerator
    {
        private int _count = 11;
        public bool MoveNext() => _count-- > 0;
        public object Current => _count;
    }
    class Program
    {
        static void Main(string[] args)
        {
            IEnumerator e = new Countdown();
            while (e.MoveNext())
            {
                Console.Write(e.Current); // 109876543210
            }
        }
    }
}
```

Примеры использования интерфейсов

Часто программистам приходится создавать собственные интерфейсы, в основном чтобы обобщить ряд похожих объектов. Рассмотрим небольшой пример программы, которая будет формировать отчеты на основе полученных данных. Форматы отчетов будут разные (2 видов), и до компиляции программы мы не сможем узнать, какой формат выберет пользователь.

Создадим 2 класса, которые описывают выгрузку данных в 2 различных форматах (Word и Excel), и интерфейс для обобщения:

```
using System;

internal interface IExport
{
    void Export();
    void Export(string path);
}

internal class ExportWord : IExport
{
    public void Export()
    {
        throw new NotImplementedException();
    }

    public void Export(string path)
    {
        throw new NotImplementedException();
    }
}
internal class ExportExcel : IExport
{
    public void Export()
    {
        throw new NotImplementedException();
    }

    public void Export(string path)
    {
        throw new NotImplementedException();
    }
}

internal class Program
{
    public static void Main()
    {
        IExport export;

        if (true) // Теперь мы можем по условию создать нужный нам объект
        {
            export = new ExportExcel();
        }
        else
        {
            export = new ExportWord();
        }
    }
}
```

Интерфейсы, как и классы, могут наследоваться:

```
using System;

internal interface IExport : IExample
{
    void Export();
    void Export(string path);
}

internal interface IExample
{
    void Test();
}

internal class ExportWord : IExport
{
    public void Export()
    {
        throw new NotImplementedException();
    }

    public void Export(string path)
    {
        throw new NotImplementedException();
    }

    public void Test()
    {
        throw new NotImplementedException();
    }
}
```

Рассмотрим случай, когда в 2 разных интерфейсах определены функции с одинаковыми именами:

```
using System;

internal interface IExportLocally
{
    void Export();
}

internal interface IExportToServer
{
    void Export();
}

internal class ExportWord : IExportLocally, IExportToServer
{
    public void Export()
    {
        throw new NotImplementedException();
    }
}

internal class Program
{
    public static void Main()
    {
        IExportLocally export = new ExportWord();
        IExportToServer export2 = new ExportWord();
        export.Export();
        export2.Export();
    }
}
```

Класс **ExportWord** определяет один метод **Export()**, создавая общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект **ExportWord** как объект типа **IExportLocally** или **IExportToServer**, результат метода будет один и тот же.

Но нередко бывает необходимо разграничить реализуемые интерфейсы. В этом случае надо явным образом применить интерфейс:

```
internal class ExportWord : IExportLocally, IExportToServer
{
    void IExportLocally.Export()
    {
        throw new NotImplementedException();
    }

    void IExportToServer.Export()
    {
        throw new NotImplementedException();
    }
}
```

При явной реализации указывается название метода вместе с названием интерфейса. При этом мы не можем использовать модификатор **public**, то есть методы являются закрытыми. В этом случае при использовании метода **Export** в программе надо привести объект к типу соответствующего интерфейса:

```
internal class Program
{
    public static void Main()
    {
        ExportWord export = new ExportWord();

        ((IExportLocally)export).Export();
        ((IExportToServer)export).Export();
    }
}
```

Стандартные интерфейсы

Интерфейс IComparable. Сортировка по одному критерию

Во многих классах приходится реализовывать интерфейс **IComparable**, поскольку он позволяет сравнивать один объект с другим, используя различные методы, определенные в среде **.NET Framework**.

Интерфейс **IComparable** реализуется чрезвычайно просто, потому что он состоит всего лишь из одного метода:

```
int CompareTo(object obj)
```

В этом методе значение вызывающего объекта сравнивается со значением объекта, определяемого параметром **obj**. Если значение вызывающего объекта больше, чем у объекта **obj**, то возвращается положительное значение; если оба значения равны – нулевое значение; если значение вызывающего объекта меньше, чем у объекта **obj** – отрицательное значение.

Интерфейс IComparer. Сортировка по разным критериям

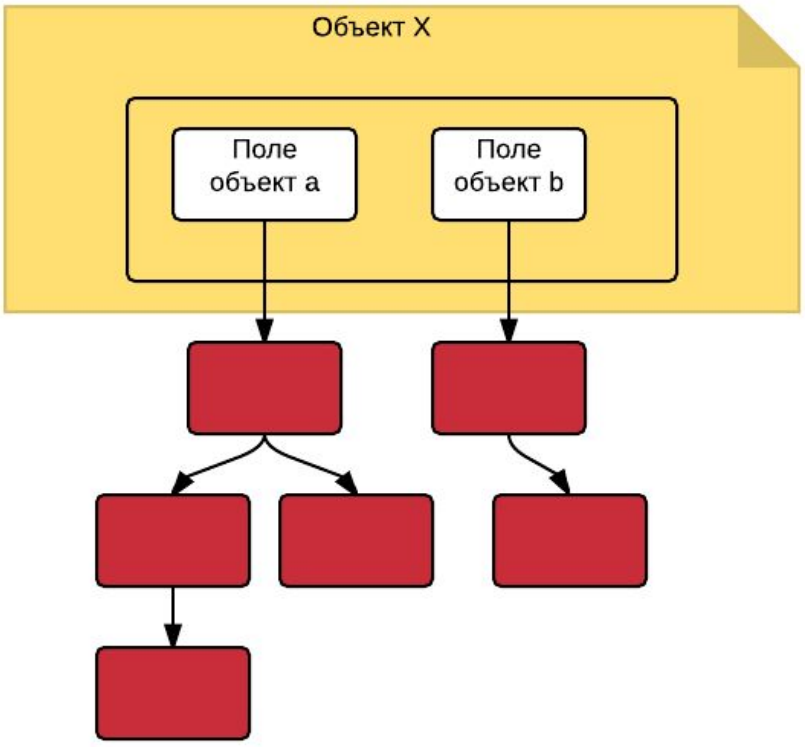
Интерфейс **IComparer** определен в пространстве имен **System.Collections**. Он содержит один метод **CompareTo**, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
Interface IComparer
{
  Int Compare (object obj1, object obj2)
}
```

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Клонирование объектов (интерфейс ICloneable)

Клонирование – это создание копии объекта (клона). При присваивании одного объекта ссылочного типа другому, копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом **MemberwiseClone**, который любой объект наследует от класса **object**. При этом объекты (ссылки), на которые указывают поля объекта, не копируются. Это называется поверхностным клонированием.



Для создания полностью независимых объектов необходимо глубокое клонирование, когда в памяти создается дубликат всего дерева объектов (тех, на которые ссылаются поля объекта, поля полей и

так далее). Алгоритм глубокого клонирования сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Пример реализации интерфейса **ICloneable** для «Астероида»:

```
using System;
using System.Drawing;

namespace MyGame
{
    class Asteroid : BaseObject, ICloneable
    {
        //...
        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));
            // Не забываем скопировать новому астероиду Power нашего астероида
            asteroid.Power = Power;
            return asteroid;
        }
        //...

    }
}
```

Рассмотрим еще один пример. Допустим, у нас есть класс для хранения информации о пользователе:

```
using System;

internal class Contact : ICloneable
{
    public string FirstName;
    public string LastName;
    public int Age;

    public object Clone()
    {
        return new Contact
        {
            FirstName = FirstName,
            LastName = LastName,
            Age = Age
        };
    }
}
```

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

```
public object Clone() => MemberwiseClone();
```

Этот метод реализует поверхностное (неглубокое) копирование, но его может быть недостаточно. Например, пусть класс **Contact** содержит ссылку на объект **Address**. Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:

```
using System;

internal class Contact : ICloneable
{
    public string FirstName;
    public string LastName;
    public int Age;
    public Adress Adress;

    public object Clone()
    {
        return new Contact
        {
            FirstName = FirstName,
            LastName = LastName,
            Age = Age,
            Adress = new Adress
            {
                City = Adress.City,
                Street = Adress.Street
            }
        };
    }
}

internal class Adress
{
    public string City;
    public string Street;
}
```

Dispose

Метод финализации может применяться для освобождения *неуправляемых* ресурсов при активизации процесса сборки мусора. Однако многие неуправляемые объекты являются «ценными элементами» (например, низкоуровневые соединения с базой данных или файловые дескрипторы). Зачастую выгоднее освобождать их как можно раньше, еще до наступления момента сборки мусора. Поэтому вместо переопределения **Finalize()** в качестве альтернативного варианта также можно реализовать в классе интерфейс **IDisposable**, который имеет единственный метод – **Dispose()**:

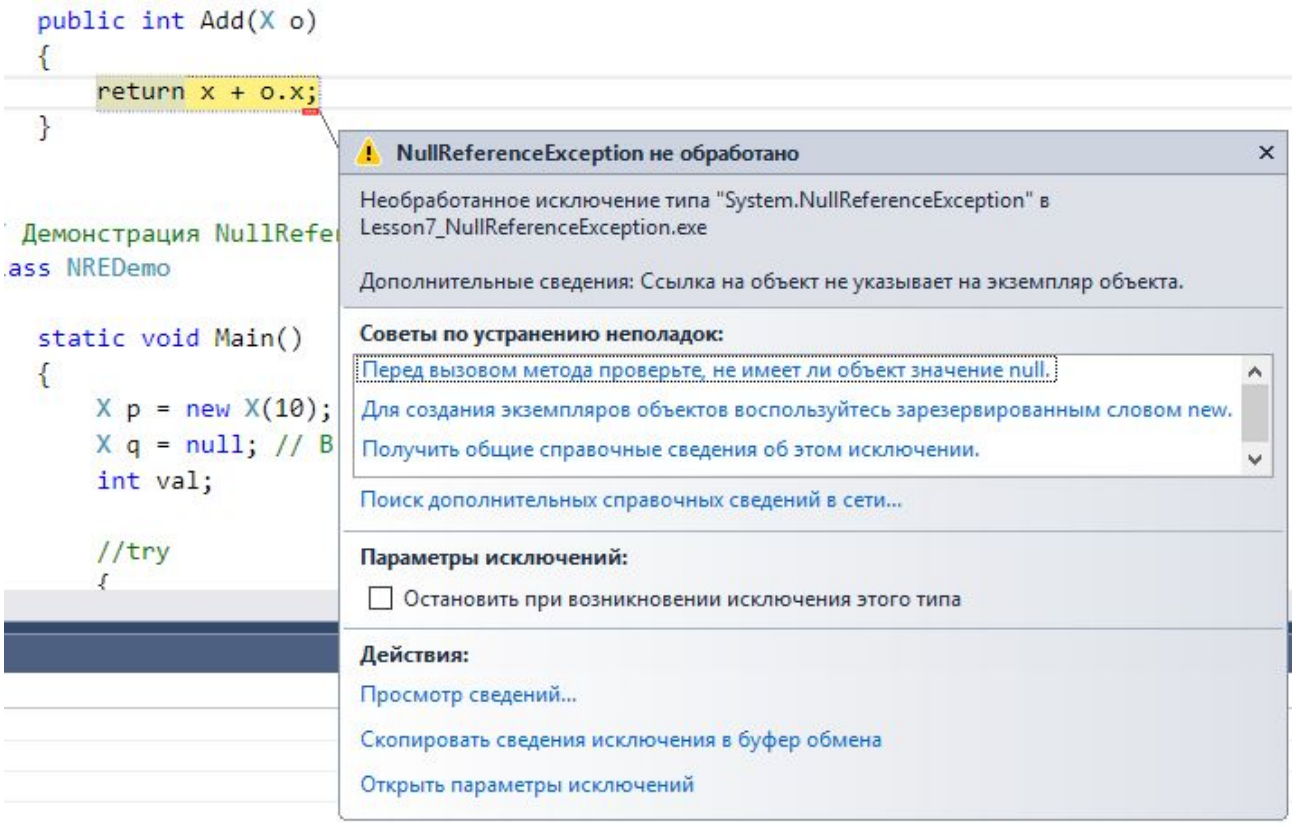
```
public interface IDisposable
{
  void Dispose();
}
```

Когда действительно реализуется поддержка интерфейса **IDisposable**, предполагается, что после завершения работы с объектом метод **Dispose()** должен вручную вызываться пользователем этого объекта прежде, чем объектной ссылке будет позволено покинуть область действия. Благодаря этому объект может выполнять любую необходимую очистку неуправляемых ресурсов без попадания в очередь финализации и без ожидания того, когда сборщик мусора запустит содержащуюся в классе логику финализации. Объекты **IDisposable**, как правило, работают в связке с конструкцией **using**.

```
public static void Main()
{
   using (StreamReader reader = new StreamReader(path))
   {
       // Работаем с объектом
   }   // Неявно вызвана функция Dispose()
       // Объект выгружен из памяти
}
```

Исключительная ситуация

Скорее всего, вы уже сталкивались с исключительными ситуациями при написании программ. В C# так называется ситуация, не предусмотренная программистом. Вот пример возникновения довольно частой исключительной ситуации – ссылка на неинициализированный объект :



Среда **Visual Studio** и язык программирования C# предоставляет богатый набор возможностей для обработки исключительных ситуаций.

Обработка исключений

Язык C#, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью механизма обработки исключений. Исключение – это объект, генерирующий информацию о «необычном программном происшествии». При этом важно различать ошибку в программе, ошибочную ситуацию и исключительную ситуацию.

Ошибка в программе допускается программистом при разработке. Например, вместо операции сравнения (==) используется операция присваивания (=). Программист должен исправить подобные ошибки до передачи кода программы заказчику. Механизм обработки исключений – это не защита от ошибок в программе.

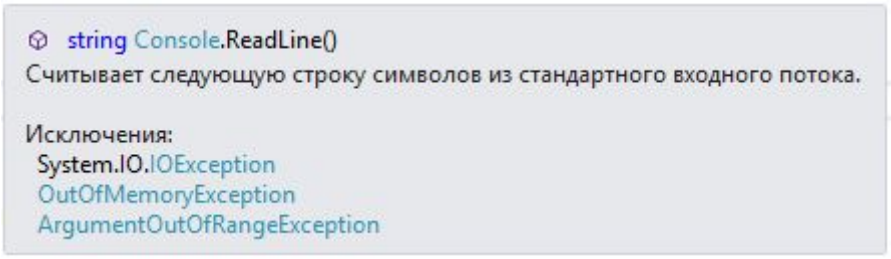
Ошибочная ситуация вызвана действиями пользователя. Например, вместо числа введена строка. Такая ошибка способна вызывать исключение. Программист должен предвидеть ошибочные ситуации и предотвращать их с помощью операторов, проверяющих допустимость поступающих данных.

Даже если программист исправил все свои баги в программе и предвидел все ошибочные ситуации, он все равно может столкнуться с непредсказуемыми и неотвратимыми проблемами –

исключительными ситуациями. Например, с нехваткой доступной памяти или попыткой открыть несуществующий файл. Исключительные ситуации разработчик предвидеть не может, но может отреагировать на них так, что они не приведут к краху программы.

Для обработки ошибочных исключительных ситуаций в C# используется специальная подсистема обработки исключений. Ее преимущество – в автоматизации создания большей части кода по обработке исключений. Раньше этот код приходилось вводить в программу вручную. Обработчик исключений также способен распознавать и выдавать информацию о таких стандартных исключениях, как деление на ноль или попадание вне диапазона определения индекса.

Visual Studio позволяет легко определить, какие исключения может выдать тот или иной метод. Для этого достаточно навести на метод мышью – и **IntelliSense** выведет описание метода со списком исключений, которые он может сгенерировать.



Рассмотрим пример перехвата исключения:

```
using System;
using System.IO;
// Пример простого перехвата исключения
// Записываем в файл данные, введенные с клавиатуры
namespace Lesson7_Exception_002
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamWriter sw=null;
            try
            {
        var path = Path.Combine(@"C:\", "temp", "text.txt");
                sw = new StreamWriter(path);
                int a;
                do
                {
                    a = Convert.ToInt32(Console.ReadLine());
                    sw.WriteLine(a);
                }
                while (a != 0);
            }
            catch (FormatException)
            {
                Console.WriteLine("Ошибка ввода данных");
            }
            catch(IOException)
            {
                Console.WriteLine("Ошибка ввода/вывода");
            }
            catch(Exception exc)
            {
                Console.WriteLine("Неизвестная ошибка");
                Console.WriteLine("Информация об ошибке" + exc.Message);
            }
            finally
            {
// Использование блока finally гарантирует, что набор операторов будет
выполняться всегда, независимо от того, возникло исключение любого типа или
нет)
                sw?.Close();
            }
        }
    }
}
```

Синтаксис оператора **try**:

```
try   // Контролируемый блок
{
...
}
catch    // Один или несколько блоков обработки исключений
{
...
}
finally   // Блок завершения
{
...
}
```

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в блок **try**. Если исключение возникает в этом блоке, оно дает о себе знать выбросом определенного рода информации. Она может быть перехвачена и обработана с помощью блока **catch**. Любой код, который должен быть обязательно выполнен при выходе из блока **try**, помещается в блок **finally**.

При необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки:

```
try
{
}
catch (FileNotFoundException e)
{
    // Обработка исключения, возникшего при отсутствии файла
}
catch (IOException e)
{
    // Обработка исключений ввода-вывода
}
catch (Exception)
{
    // Остальные исключения
}
```

Если возникает исключение определенного типа, оно переходит к соответствующему блоку **catch**. При этом более частные исключения следует помещать в начале, и только потом – более общие классы исключений. Например, сначала обрабатывается исключение **IOException**, и затем **Exception** (так как **IOException** наследуется от класса **Exception**).

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. Так мы сможем пробросить исключение дальше, до ближайшего **try catch**:

```
string password = "1234";
if (password.Length < 6)
{
    throw new Exception("Длина пароля меньше 6 символов");
}
```

```
int x = Int32.Parse("2");
x *= x;
Console.WriteLine("Квадрат числа: " + x);
```

Если пользователь введет не число, а строку или некорректные символы, то программа выдаст ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок **try catch**, чтобы обработать возможную ошибку. Но оптимально было бы проверить допустимость преобразования:

```
string input = "7";
if (Int32.TryParse(input, out var x)) // С версии C# 6.0 мы можем объявлять
переменные прямо при передаче в параметр
{
    x *= x;
    Console.WriteLine("Квадрат числа: " + x);
}
else
{
    Console.WriteLine("Некорректный ввод");
}
```

Метод **Int32.TryParse()** возвращает **true**, если преобразование можно осуществить, и **false** – если нельзя. При допустимости преобразования переменная **x** будет содержать введенное число. Так, не используя **try catch**, можно обработать возможную исключительную ситуацию. С точки зрения производительности использование блоков **try catch** более накладно, чем применение условных конструкций. Поэтому по возможности вместо **try catch** лучше использовать условные конструкции на проверку исключительных ситуаций.

В C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как фильтры исключений. Они позволяют обрабатывать исключения в зависимости от определенных условий:

```
int x = 1;
int y = 0;
try
{
    int result = x / y;
}
catch (Exception ex) when (y == 0)
{
    Console.WriteLine("y не должен быть равен 0");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Пример сокрытия ошибок с помощью перехвата исключения

В примере будем делить элементы массива **numer** на элементы массива **denum**. При делении на 0 будет возникать исключение, но мы будем его перехватывать и продолжать выполнение программы:

```
// Изящный способ сокрытия ошибок
// Из книги Герберта Шилдта «C# 4.0. Полное руководство»  (2011 г.)
using System;
class ExcDemo3 {
  static void Main() {
    int[] numer = { 4, 8, 16, 32, 64, 128 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };
    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " is " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException e) {
// Перехват исключения
        Console.WriteLine("Делить на 0 нельзя");
      }
    }
  }
```

Пример исправления нулевой ссылки при помощи перехвата исключения

Создадим ситуацию с попыткой обратиться к несозданному объекту. Продемонстрируем, как можно исправить ошибку:

```
// Использование NullReferenceException
// Из книги Герберта Шилдта «C# 4.0. Полное руководство»  (2011 г.)
using System;
class X
{
    int x;
    public X(int a)
    {
        x = a;
    }
    public int Add(X o)
    {
        return x + o.x;
    }
}
// Демонстрация NullReferenceException.
class NREDemo
{
    static void Main()
    {
        X p = new X(10);
        X q = null;                 // В q специально присваиваем null
        int val;
        try
        {
// if (q == null) q = new X(10);
                val = p.Add(q); // Обращение приведет к исключению
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException!");
            Console.WriteLine("исправляем...\n");
// Теперь исправим
            q = new X(9);
            val = p.Add(q);
        }
        Console.WriteLine("Значение {0}", val);
    }
}
```

Генерация собственных исключений

До сих пор мы рассматривали исключения, которые генерирует среда, но сгенерировать исключение может и сам программист. Для этого необходимо воспользоваться оператором **throw**, указав параметры, определяющие вид исключения. Параметром должен быть объект, порожденный от стандартного класса **System.Exception**. Этот объект используется для передачи обработчику информации об исключении:

```
static void Main()
      {
            try
            {
            int x = int.Parse(Console.ReadLine());
            if (x < 0) throw new Exception();
                Console.WriteLine("ok");
            }
            catch
            {
                Console.WriteLine("введено недопустимое значение");
            }
      }
```

Создание собственных исключений:

```
// Создание собственного исключения
// Из книги Герберта Шилдта «C# 4.0. Полное руководство»  (2011 г.)
using System;
// Для собственного исключения создаем класс, производный от класса Exception
class MyException : Exception
{
    public MyException()
    {
        Console.WriteLine(base.Message);
    }
}
class ThrowDemo
{
    static void Main()
    {
//  DateTime date = new DateTime(2016, 13, 40);
        try
        {
            Console.WriteLine("До возникновения исключения.");
            throw new MyException();
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Перехват исключения.");
        }
        catch (MyException)
        {
            Console.WriteLine("Сработало мое исключение!.");
        }
        Console.WriteLine("После блока обработки try/catch.");
    }
}
```

Вот еще один пример создания собственного исключения и его выброса:

```
using System;
namespace Lesson7_HW3
{
    class RobotException :Exception
    {
        public RobotException(string message)
            : base(message)
        {
        }
    }
    class Robot
    {
        const int MAX_HEIGHT=100, MAX_WIDTH=100;
        int height, width;
        public Robot(int высота, int ширина)
        {
            if (высота > MAX_HEIGHT) throw new RobotException("Превышена
максимальная высота");
            if (высота > MAX_WIDTH) throw new RobotException("Превышена
максимальная ширина");
            height = ширина;
            width = высота;
            Console.WriteLine("Робот построен");
        }
    }
}
```

Советы по работе с исключениями

Несколько общих рекомендаций по обработке исключений:

- Платформа **.NET Framework** активно использует механизм исключений для уведомлений об ошибках и их обработки. Поступайте так же.
- Тем не менее, исключения предназначены для индикации исключительных ситуаций, а не для контроля за ходом выполнения программы. Если объект не может принимать значение **null**, выполняйте простую проверку сравнением, не перекладывая работу на исключение. То же самое относится к делению на ноль и ко многим другим простым ошибкам.
- Исключения стоит применять лишь в крайних ситуациях еще и потому, что они расходуют много памяти и времени.
- Исключения должны содержать максимум полезной информации, помогающей в диагностике и решении проблемы (с учетом предостережений, приведенных ниже).
- Не показывайте необработанные исключения пользователю. Их следует регистрировать в журнале, чтобы разработчики смогли впоследствии устранить проблему.
- Будьте осторожны в раскрытии информации. Помните, что злонамеренные пользователи могут извлечь из исключений информацию о том, как работает программа и какие уязвимости она имеет.

- Не перехватывайте корневой объект всех исключений **system.Exception**. Он поглотит все ошибки, которые необходимо проанализировать и исправить. Это исключение хорошо перехватывать в целях регистрации, если вы собираетесь возбудить его повторно.

- Помещайте исключения низкого уровня в свои исключения, чтобы скрыть детали реализации. Например, если у вас есть коллекция, реализованная с помощью **List<T>**, имеет смысл скрыть исключение **ArgumentOutOfRangeException** внутри исключения **MyComponentException**.

Практика

«Астероид» с использованием интерфейсов

Для определения столкновений опишем интерфейс **ICollision**. Он закладывает поведение, по которому два объекта, поддерживающие его, могут определить, столкнулись ли они. Для определения столкновения используем метод **IntersectsWith** структуры **Rect**:

```
using System.Drawing;
interface ICollision
{
    bool Collision(ICollision obj);
    Rectangle Rect { get; }
}
```

Теперь нужно наследовать этот интерфейс объектами, которые могут столкнуться. Проще наследовать и реализовывать этот интерфейс в базовом классе, тогда **Asteroid** и **Bullet** будут поддерживать поведение обнаружения столкновений.

Базовый класс **BaseObject**, который наследует и реализовывает **ICollision**:

```
using System;
using System.Drawing;

namespace MyGame
{
    abstract class BaseObject : ICollision
    {
        protected Point Pos;
        protected Point Dir;
        protected Size Size;

        protected BaseObject(Point pos, Point dir, Size size)
        {
            Pos = pos;
            Dir = dir;
            Size = size;
        }
        public abstract void Draw();
        public virtual void Update()
        {
            Pos.X = Pos.X + Dir.X;
            if (Pos.X < 0) Pos.X = Game.Width + Size.Width;
        }

        // Так как переданный объект тоже должен будет реализовывать интерфейс
ICollision, мы
        // можем использовать его свойство Rect и метод IntersectsWith для
обнаружения пересечения с
        // нашим объектом (а можно наоборот)
        public bool Collision(ICollision o) => o.Rect.IntersectsWith(this.Rect);

        public Rectangle Rect => new Rectangle(Pos, Size);

    }
}
```

Теперь добавим обнаружение столкновений в класс **Game** в метод **Update** и перепишем метод Draw:

```
public static void Draw()
{
    Buffer.Graphics.Clear(Color.Black);
    foreach (BaseObject obj in _objs)
        obj.Draw();
    foreach (Asteroid obj in _asteroids)
        obj.Draw();
    _bullet.Draw();
    Buffer.Render();
}

public static void Update()
{
    foreach (BaseObject obj in _objs)
        obj.Update();
    foreach (Asteroid a in _asteroids)
    {
        a.Update();
        if (a.Collision(_bullet)) { System.Media.SystemSounds.Hand.Play(); }
    }
    _bullet.Update();
}
```

Обозначим столкновение простым системным звуком. Запустите программу и убедитесь, что при столкновении снаряда с астероидом проигрывается звук.

Код класса **Program**:

```
using System;
using System.Windows.Forms;
// Создаем шаблон приложения, где подключаем модули
namespace MyGame
{
    class Program
    {
        static void Main(string[] args)
        {
            Form form = new Form
            {
                Width = Screen.PrimaryScreen.Bounds.Width,
                Height = Screen.PrimaryScreen.Bounds.Height
            };
            Game.Init(form);
            form.Show();
            Game.Load();
            Game.Draw();
            Application.Run(form);
        }
    }
}
```

Примеры

Пример использования абстрактного класса и абстрактного метода в задаче вывода значений некоторой функции:

```
// Универсальный метод вывода таблицы значений функции можно реализовать с помощью
// абстрактного базового класса, содержащего два метода: метод вывода таблицы и абстрактный
// метод, задающий вид вычисляемой функции
namespace AbstractClass {
    abstract class TableFun
    {
        public abstract double F(double x);
        public void Table(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("----------------------");
        }
    }
    class SimpleFun : TableFun
    {
        public override double F(double x)
        {
            return x * x;
        }
    }
    class SinFun : TableFun
    {
        public override double F(double x)
        {
            return Math.Sin(x);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            TableFun a = new SinFun();
            Console.WriteLine("Таблица функции Sin:");
            a.Table(-2, 2);
            a = new SimpleFun();
            Console.WriteLine("Таблица функции Simple:");
            a.Table(0, 3);
        }
    }
}
```

Как научить foreach работать с вашими данными?

```
using System;
using System.Collections;
// Пример необходимости реализации интерфейсов IEnumerable и IEnumerator
// Здесь показано, как научить foreach работать с вашими данными
// Для циклического обращения к элементам коллекции зачастую проще (да и
лучше) организовать
// цикл foreach, чем пользоваться непосредственно методами интерфейса
IEnumerator
// Если требуется создать класс, содержащий объекты, перечисляемые в цикле
foreach, то
// в этом классе следует реализовать интерфейсы IEnumerator и IEnumerable
// Чтобы обратиться к объекту определяемого пользователем класса в цикле
foreach,
// необходимо реализовать интерфейсы IEnumerator и IEnumerable в их обобщенной
или
// необобщенной форме.
namespace Interfaces_060_MyClass_and_Foreach
{
	class MyClass : IEnumerable, IEnumerator
	{
		// Наш пользовательский тип данных
		private readonly int[] _a;

		public MyClass(int n)
		{
			_a = new int[n];
		// Заполняем его произвольными данными
			for (var i = 0; i < n; i++) _a[i] = i;
		}
		// Первоначально индекс указывает на -1, так как к переходу к
следующему мы увеличим его на 1
		private int _i = -1;
		// Реализуем интерфейс IEnumerable
		// Этот интерфейс должен только вернуть объект типа IEnumerator,
который будет заниматься перечислением элементов
		//-----------------------------------------------
		public IEnumerator GetEnumerator()
		{
			return this;
		}
		//-----------------------------------------------
		// Теперь реализуем интерфейс IEnumerator
		//-------------------------------------
		public bool MoveNext()
		{
			if (_i == _a.Length - 1)
			{
				Reset();
				return false;
			}
```

```
                _i++;
                return true;
            }
            public void Reset()
            {
                _i = -1;
            }
            public object Current => _a[_i];
        }
        //------------------------------------------------------------
        class Program
        {
            static void Main(string[] args)
            {
                MyClass my = new MyClass(5);
                foreach (var m in my)
                    Console.Write("{0,4}", m);
            }
        }
}
```

Реализация интерфейса IEnumerable с использованием ключевого слова yield

```
using System;
using System.Collections;                // Необходим для интерфейса
IEnumerable
                                         // Использование итератора
namespace Interfaces_060_MyClass_and_Foreach_2
{
    class MyClass: IEnumerable
    {
       private readonly int[] _mass;
        public MyClass(int n)
        {
            _mass= new int[n];
            for (int i = 0; i < n; i++) _mass[i] = i;
        }
        public IEnumerator GetEnumerator()
        {
            for (int i = 0; i < _mass.Length; i++)
                yield return _mass[i];
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyClass mc = new MyClass(10);
            foreach (var m in mc)
                Console.WriteLine(m);
        }
    }
}
```

Пример загрузки данных в класс с массивом и сортировка через реализацию IComparable

На вход программе подаются сведения о сдаче экзаменов учениками 9 классов средней школы. В первой строке сообщается количество учеников **N**, которое не меньше 10, но не превосходит 100. Каждая из следующих N-строк имеет следующий формат:

```
<Фамилия> <Имя> <оценки>
```

<Фамилия> – строка, состоящая не более чем из 20 символов, **<Имя>** – строка, состоящая не более чем из 15 символов, **<оценки>** – через пробел три целых числа, соответствующие оценкам по пятибалльной системе. **<Фамилия>** и **<Имя>**, а также **<Имя>** и **<оценки>** разделены одним пробелом. Пример входной строки:

```
Иванов Петр 4 5 3
```

Требуется написать как можно более эффективную программу (укажите используемую версию языка программирования – например, Borland Pascal 7.0), которая будет выводить на экран фамилии и имена трех худших по среднему баллу учеников. Если среди остальных есть ученики, набравшие тот же средний балл, что и один из трех худших, следует вывести и их фамилии и имена.

Для выполнения задачи нужно создать файл с данными **data.txt**:

```
10
Grishin Mikhei` 22 6 7
Evstaf`ev Iulian 5 23 1
Vakhrov Aggei` 19 13 20
Degtiarev Savvatii` 8 12 23
Noskov Ul`ian 7 4 1
Vennikov Venedikt 3 17 16
Masharin Demid 22 16 7
Biriuk Andronik 2 7 21
Chernakov Vitalii` 18 7 8
Climov Sviatopolk 24 17 11
```

```
using System;
using System.IO;
namespace HW_TaskEGE
{
      class Element : IComparable
      {
            private readonly string _fio;
            private readonly int _ball;
            public Element(string fio, int ball)
            {
                  _fio = fio;
                  _ball = ball;
            }
            public string FIO => _fio;
```

```
            public int Ball => _ball;

            public int CompareTo(object obj)
            {
                  if (_ball < ((Element) obj).Ball) return 1;
                  if (_ball > ((Element) obj).Ball) return -1;
                  return 0;
            }
      }
      class Program
      {
            static void Main(string[] args)
            {
                  StreamReader sr = new StreamReader("data.txt");
                  int n = Int32.Parse(sr.ReadLine());
                  Element[] list = new Element[n];
                  for (var i = 0; i < n; i++)
                  {
                        string[] s = sr.ReadLine().Split(' ');
                        int ball = Int32.Parse(s[2]) + Int32.Parse(s[3]) +
Int32.Parse(s[4]);
                        list[i] = new Element(s[0] + " " + s[1], ball);
                  }
                  sr.Close();
                  Array.Sort(list);
                  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);
                  }
            }
      }
}
```

Перехват исключений. Использование блока finally

```
using System;
// Пример перехвата исключения в зависимости от типа
class Program
{
    static void Main(string[] args)
    {
        System.IO.StreamWriter sw=null;
        try
        {
            sw = new System.IO.StreamWriter("data.txt");
            Console.WriteLine("Введите напряжение:");
            int u = int.Parse(Console.ReadLine());
            Console.WriteLine("Введите сопротивление:");
            int r = int.Parse(Console.ReadLine());
            int i = u / r;
            Console.WriteLine("Сила тока - " + i);
            sw.WriteLine("При напряжении {0} и сопротивлении {1} сила тока:", u,
r, i);
            sw.Close();
        }
        catch (FormatException e)
        {
            Console.WriteLine("Неверный формат ввода! " + e.Message);
            return;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Деление на 0!");
        }
        catch(Exception e) // Общий случай
        {
            Console.WriteLine("Неопознанное исключение"+e);
        }
        finally
        {
            if (sw!=null) sw.Close();
            Console.WriteLine("Закрываем используемые ресурсы");
            Console.ReadKey();
        }
    }
}
```

Домашнее задание

1. Построить три класса (базовый и 2 потомка), описывающих работников с почасовой оплатой (один из потомков) и фиксированной оплатой (второй потомок):
    a. Описать в базовом классе абстрактный метод для расчета среднемесячной заработной платы. Для «повременщиков» формула для расчета такова:

«среднемесячная заработная плата = 20.8 * 8 * почасовая ставка»; для работников с фиксированной оплатой: «среднемесячная заработная плата = фиксированная месячная оплата»;

b. Создать на базе абстрактного класса массив сотрудников и заполнить его;

c. * Реализовать интерфейсы для возможности сортировки массива, используя **Array.Sort()**;

d. * Создать класс, содержащий массив сотрудников, и реализовать возможность вывода данных с использованием **foreach**.

2. Переделать виртуальный метод **Update** в **BaseObject** в абстрактный и реализовать его в наследниках.
3. Сделать так, чтобы при столкновении пули с астероидом они регенерировались в разных концах экрана.
4. Сделать проверку на задание размера экрана в классе **Game**. Если высота или ширина (Width, Height) больше 1000 или принимает отрицательное значение, выбросить исключение **ArgumentOutOfRangeException()**.
5. * Создать собственное исключение **GameObjectException**, которое появляется при попытке создать объект с неправильными характеристиками (например, отрицательные размеры, слишком большая скорость или неверная позиция).

Дополнительные материалы

1. yield (справочник по C#)

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. Бен Ватсон. C# 4.0 на примерах. – 2010 г.
5. MSDN.