Урок 2

Объектно - Ориентированное Программирование

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

Абстрактный метод

Абстрактный метод не реализуется в базовом классе. Он должен быть переопределён в наследуемом классе. Абстрактными могут быть также индексаторы и свойства

```
abstract class MyClass
  {
     abstract protected 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. Эти классы не могут использоваться для создания объектов, потому что абстрактные классы не завершены - производные классы должны предоставить «недостающие части».

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

На первый взгляд может показаться очень странным, зачем определять класс, экземпляр которого нельзя создать непосредственно. Однако вспомните, что базовые классы (абстрактные или нет) очень полезны тем, что содержат общие данные и общую функциональность для унаследованных типов. Используя эту форму абстракции, можно также моделировать общую “идею”, а не обязательно конкретную сущность. Также следует понимать, что 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 использование абстрактного класса

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

```
abstract class BaseObject
```

Добавим слово abstract перед названием метода Draw. Virtual нужно убрать, так как абстрактный метод в общем-то и так виртуальный. Удалите тело метода Draw.

```
abstract class BaseObject
{
    protected Point pos;
    protected Point dir;
    protected Size size;
    public BaseObject(Point pos, Point dir, Size size)
    {
        this.pos = pos;
        this.dir = dir;
        this.size = size;
    }
    abstract public void Draw();
    virtual public 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 даёт возможность задавать некоторую 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 и индексаторы.

Создадим ещё один класс Asteroid, который будет реализовывать метод Draw.

```
// Создаём 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 объекта класса Star, Asteroid и Bullet.

Объекты bullet и asteroid должны быть описаны в классе Game.

```
static Bullet bullet;
static Asteroid[] asteroids;
static public 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];
   for (int i = 0; i < objs.Length; i++)
   {
      int r = rnd.Next(5, 50);
     objs[i] = new Star(new Point(1000, Game.rnd.Next(0, Game.Height)), new Point(-r, r), new Size(3, 3));
    }
    for (int i = 0; i < asteroids.Length; i++)
    {
      int r = rnd.Next(5, 50);
      asteroids[i] = new Asteroid(new Point(1000, Game.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 описаны ниже.

- Члены интерфейса всегда неявно являются абстрактными. В противоположность этому класс может содержать как абстрактные, так и конкретные методы с реализацией.

- Класс (или структура) может реализовать несколько интерфейсов. Однако 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 являются абстрактными. 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, который может быть абстрактным.

Рассмотрим немного упрощённую версию интерфейса IEnumerator, определённую в классе System. Collections,

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

Оператор foreach применяется только к классам, в которых реализован интерфейс INumerator. Этот вроде бы простой пример демонстрирует важность интерфейсов. Интерфейсы задают только то, что должен выполнять класс, но не говорят, как это делается. Скажем, 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 являются открытыми и не могут объявлять модификатор доступа. Реализация интерфейса означает открытую реализацию всех его членов.

```
class Countdown : IEnumerator
{
      int count = 11;
      public bool MoveNext()
      {
         return count-- > 0 ;//Здесь count-- и сравнение получившегося значения с 0
      }
      public object Current { get { return 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
    {
        int count = 11;
        public bool MoveNext() { return count-- > 0 ; }
        public object Current { get { return count; } }
    }
    class Program
    {
        static void Main(string[] args)
        {
        IEnumerator e = new Countdown();
            while (e.MoveNext())
                Console.Write (e.Current); // 109876543210
        }
    }
}
```

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, определённые 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, а если значение вызывающего объекта меньше, чем у объекта obj, отрицательное значение.

Интерфейс IComparer. Сортировка по разным критериям

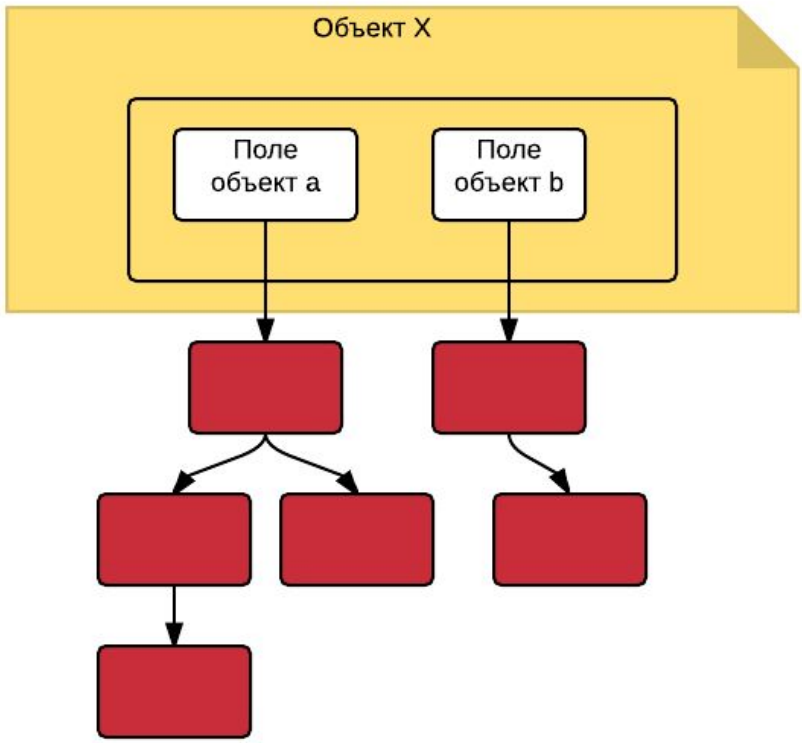
Интерфейс IComparer определён в пространстве имён System.Collections. Он содержит один метод CompareTo, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
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 передаётся в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Клонирование объектов (интерфейс ICloneable)

Клонирование - это создание копии объекта. Копия объекта называется клоном. Как вам известно, при присваивании одного объекта ссылочного типа другому, копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объект, можно воспользоваться методом MemberwiseClone, который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется поверхностным клонированием.



Для создания полностью независимых объектов необходимо глубокое клонирование, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей и т.д. Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Пример реализации интерфейса ICloneable для астероида:

```
   class Asteroid:ICloneable
   {
           //...
       public object Clone()
       {
          // Создаём нового робота, 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;
       }
}
```

Dispose

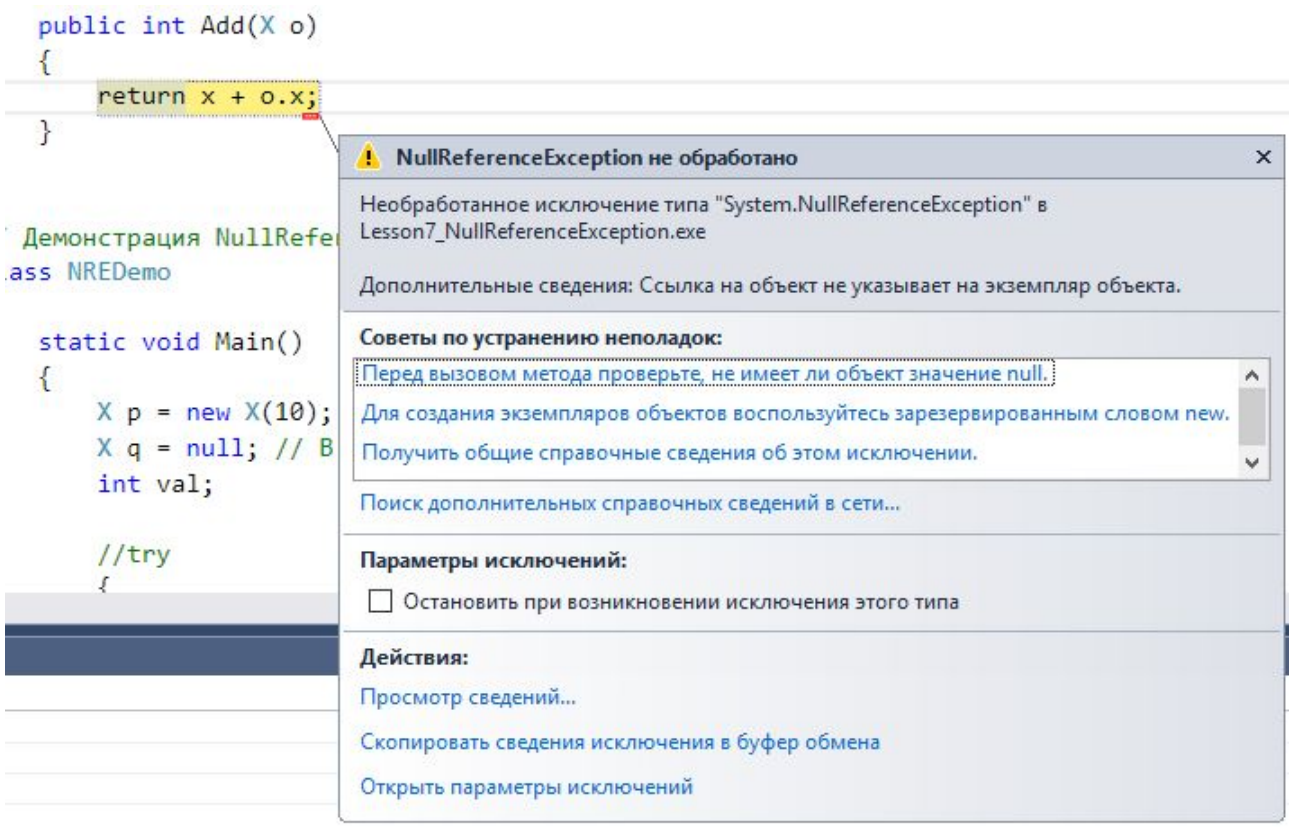
Методы финализации могут применяться для освобождения неуправляемых ресурсов при активизации процесса сборки мусора. Однако многие неуправляемые объекты являются "ценными элементами" (например, низкоуровневые соединения с базой данных или файловые дескрипторы) и часто выгоднее освобождать их как можно раньше, ещё до наступления момента сборки мусора. Поэтому вместо переопределения Finalize() в качестве альтернативного варианта также можно реализовать в классе интерфейс IDisposable, который имеет единственный метод по имени Dispose():

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

Когда действительно реализуется поддержка интерфейса IDisposable, то 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.

Исключительная ситуация

Скорее всего вы уже сталкивались с исключительными ситуациями при написании программ ранее. Исключительной ситуацией в C# называется ситуация, не предусмотренная программистом. Вот пример возникновения довольно частой исключительной ситуации ссылка на неинициализированный объект.



Среда Visual Studio и язык программирования C# предоставляет нам 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 проводить различие между ошибкой в программе, ошибочной ситуацией и исключительной ситуаций.

Ошибка в программе допускается программистом при её разработке. Например, вместо операции сравнения (==) используется операция присваивания (=). Программист должен исправить подобные ошибки до передачи кода программы заказчику. Использование механизма обработки исключений не является защитой от ошибок в программе.

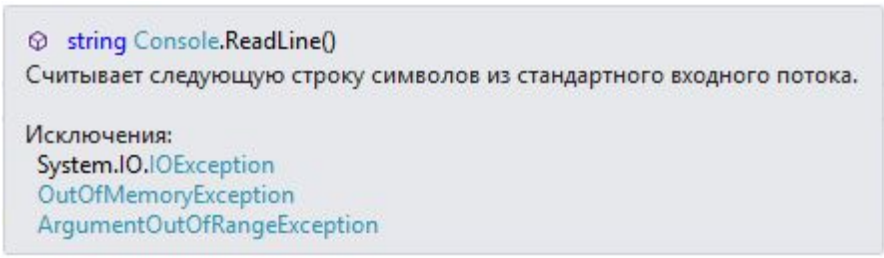
Ошибочная ситуация вызвана действиями пользователя. Например, пользователь вместо числа ввёл строку. Такая ошибка способна вызывать исключение. Программист должен предвидеть ошибочные ситуации и предотвращать их с помощью операторов, проверяющих допустимость поступающих данных.

Даже если программист исправил все свои ошибки в программе, предвидел все ошибочные ситуации, он все равно может столкнуться с непредсказуемыми и неотвратимыми проблемами - исключительными ситуациями. Например, нехваткой доступной памяти или попыткой открыть несуществующий файл. Исключительные ситуации программист предвидеть не может, но он может отреагировать на них так, что они не приведут к краху программы.

Для обработки ошибочных и исключительных ситуаций в C# используется специальная подсистема обработки исключений. Преимущество данной подсистемы состоит в автоматизации создания большей части кода по обработке исключений. Раньше этот код приходилось вводить в программу "вручную".

Кроме этого обработчик исключений способен распознавать и выдавать информацию о таких стандартных исключениях, как деление на нуль или попадание вне диапазона определения индекса.

Visual Studio позволяет нам 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 может сгенерировать этот метод.



Рассмотрим пример перехвата исключения:

```
using System;
using System.IO;
// Пример простого перехвата исключения
// Записываем в файл данные, введённые с клавиатуры
namespace Lesson7_Exception_002
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamWriter sw=null;
            try
            {
                sw = new StreamWriter("C:\\temp\\text.txt");
                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 некоторый набор операторов будет
выполняться всегда, независимо от того, возникло исключение (любого типа) или нет.
                if (sw != null) 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 даёт знать о себе выбросом определённого рода информации. Выброшенная информация может быть перехвачена и обработана соответствующим образом с помощью блока catch. Любой код, который должен быть обязательно выполнен при выходе из блока try, помещается в блок finally.

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
// C# 4.0 Полное руководство  Шилдт.Г   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 попытки обратиться к несозданному объекту. Продемонстрируем, как можно исправить ошибку.

```
// Использование NullReferenceException
// C# 4.0 Полное руководство  Шилдт.Г   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 приведёт 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, порождённый от стандартного класса System.Exception. Этот объект используется для передачи информации об исключении обработчику.

```
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
// C# 4.0 Полное руководство  Шилдт.Г   2011 г.
using System;
// Для создания собственного исключения создаём 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 ещё один пример создания собственного исключения и выброса его:

```
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

Вот несколько общих рекомендаций по обработке исключений.

- Платформа .NET Framework активно использует механизм исключений для уведомлений об ошибках и их обработки. Поступайте также.
- И тем на менее, исключения предназначены для индикации исключительных ситуаций, а не для контроля за ходом выполнения программы. Например, если объект не может принимать значение null, выполняйте простую проверку сравнением, не перекладывая работу на исключение. Тоже самое относится к делению на ноль и ко многим другим простым ошибкам.
- Одним из важных соображений в пользу применения исключений лишь в крайних ситуациях является их дороговизна в смысле расхода памяти и времени.
- Исключения должны содержать максимум полезной информации, помогающей в диагностике и решении проблемы (с учётом предостережений, приведённых 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. Этот интерфейс закладывает поведение, по которому два объекта могут определить, столкнулись ли они, если оба поддерживают интерфейс ICollision. Для определения столкновения используем метод IntersectsWith структуры Rect

```
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 и Asteroid и Bullet будут поддерживать поведение обнаружения столкновений.

Базовый класс BaseObject, который наследует и реализовывает ICollision

```
abstract class BaseObject: ICollision
{
   protected Point pos;
   protected Point dir;
   protected Size size;
   public BaseObject(Point pos, Point dir, Size size)
   {
      this.pos = pos;
      this.dir = dir;
      this.size = size;
   }
   abstract public void Draw();
   virtual public 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)
   {
      if (o.Rect.IntersectsWith(this.Rect)) return true; else return false;
   }
   public Rectangle Rect
   {
     get { return new Rectangle(pos, size); }
   }
}
```

Теперь добавим обнаружение столкновений в класс Game в метод Update, и перепишем метод Draw.

```
static public 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();
  }
static public 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.

Примеры

Пример использования абстрактного класса и абстрактный 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.Тем не менее //
ясное представление о принципе действия подобных интерфейсов важно иметь по еще одной
// причине: если требуется создать класс, содержащий объекты, перечисляемые в цикле foreach, то //
в этом классе следует реализовать интерфейсы IEnumerator и IEnumerable. Иными словами, для
// того, чтобы обратиться к объекту определяемого пользователем класса в цикле foreach,
// необходимо реализовать интерфейсы IEnumerator и IEnumerable в их обобщенной или
// необобщённой форме. Правда, сделать это будет нетрудно, поскольку оба интерфейса не очень
// велики.
namespace Interfaces_060_MyClass_and_Foreach
{
   class MyClass: IEnumerable, IEnumerator
   {
// Наш пользовательский тип данных
      int[] a;

      public MyClass(int n)
      {
         a = new int[n];
// Заполняем его произвольными данными
         for (int i = 0; i < n; i++) a[i] = i;
      }
// Первоначально индекс указывает на -1, так как к переходу к следующему мы увеличим его на 1
      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
      {
         get
```

```
            {
                return 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 INumerable 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
    {
        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-х классов некоторой 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>
```

где <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 трёх худших по среднему баллу учеников. Если среди остальных есть ученики, набравшие тот же средний балл, что и один из трёх худших, то следует вывести и их фамилии и имена.

Для выполнения задачи нужно создать файл с данными data.txt, например, такой:

```
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
   {
      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; }
      }
      public int CompareTo(object obj)
      {
```

```
            if (this.ball < (obj as Element).Ball) return 1;
            if (this.ball > (obj as Element).Ball) return -1;
            return 0;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader sr = new StreamReader("data.txt");
            int N = int.Parse(sr.ReadLine());
            Element[] list = new Element[N];
            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[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 некоторых работников с почасовой оплатой (один из потомков) и фиксированной оплатой (второй потомок).
   а) Описать в базовом классе абстрактный метод для расчёта среднемесячной заработной платы. Для «повременщиков» формула для расчета такова: «среднемесячная заработная плата = 20.8 * 8 * почасовую ставку», для работников с фиксированной оплатой «среднемесячная заработная плата = фиксированной месячной оплате».
   б) Создать на базе абстрактного класса массив сотрудников и заполнить его.
   в) **Реализовать интерфейсы для возможности сортировки массива используя Array.Sort().
   г) ***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 столкновениях пули с астероидом пуля и астероид регенерировались в разных концах экрана;
4. Сделать проверку на задание размера экрана в классе Game. Если высота или ширина больше 1000 или принимает отрицательное значение, то выбросить исключение ArgumentOutOfRangeException().
5. *Создать собственное исключение GameObjectException, которое появляется при попытке создать объект с неправильными характеристиками (например, отрицательные размеры, слишком большая скорость или позиция).

Дополнительные материалы

1. yield (справочник по C#)

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. Бен Ватсон “C# 4.0 на примерах”, БХВ-Петербург, 2010
5. MSDN.