ЛЕКЦИЯ 6

- Модуль 2
- 15.11.2023
- Абстрактные классы
- Разработка собственных типов исключений

ЦЕЛИ ЛЕКЦИИ

- Изучить абстрактные классы и их реализацию в C#
- Разобраться с описанием собственных типов исключений
- Посмотреть соглашения о стиле кодирования, связанного с исключениями

АБСТРАКТНЫЕ КЛАССЫ

АБСТРАКЦИЯ ДАННЫХ (НАПОМИНАНИЕ)

- Абстракция данных позволяет рассматривать данные с проблемно-ориентированной позиции, не затрагивая, например, вопросов их представления в памяти компьютера

Абстракция данных основывается на абстракции процесса, так как важны не столько сами данные, сколько операции, которые можно над ними выполнять

АБСТРАКТНЫЙ КЛАСС

Абстрактный класс - это тип класса, который не может быть инстанцирован напрямую, то есть вы не можете создать его объект

- основное назначение - быть базовым классом для других классов
- абстрактные классы обычно используются для создания общей, обобщенной структуры объекта, которой соответствуют конкретные классы

КЛ. СЛОВО

↓

```
abstract class MyClass  {
        ...
}
```

МОДИФИКАТОР ABSTRACT

- используется с
  - классами
  - методами
  - свойствами
  - индексаторами
  - событиями

кл. слово ↓ — точка с запятой вместо реализации ↓

```
abstract public void PrintStuff(string s);
abstract public int MyProperty      {
        get; ← точка с запятой вместо реализации
        set; ← точка с запятой вместо реализации
}
```

АБСТРАКТНЫЙ КЛАСС

- Создавать экземпляры абстрактного класса нельзя
- Абстрактный класс может содержать абстрактные члены
  - Все абстрактные члены должны быть объявлены открытыми (public)
  - Все абстрактные члены по умолчанию виртуальны
- Абстрактный класс может содержать неабстрактные члены
- Абстрактный класс не может иметь модификатор `sealed` или `static`
- Неабстрактный класс-наследник от абстрактного класса, должен включать реализации всех наследуемых абстрактных членов
  - Если он не содержит всех реализаций, то также объявляется с модификатором `abstract`
- Абстрактный класс нельзя унаследовать структурой

НАЗНАЧЕНИЕ АБСТРАКТНЫХ КЛАССОВ

- Абстрактные базовые классы можно использовать для объявления ссылок, которые могут обращаться к экземплярам многих конкретных классов, производных от абстрактного класса
- Программы могут полиморфно использовать такие ссылки для манипулирования экземплярами производных классов

АБСТРАКТНЫЙ МЕТОД

Абстрактные методы - это методы, которые объявляются в абстрактном классе без какой-либо реализации

- абстрактный метод по умолчанию является виртуальным
- объявления абстрактных методов допускаются только в абстрактных классах
- у абстрактного метода отсутствует тело, а описание заканчивается точкой с запятой

В объявлении абстрактного метода нельзя использовать статические (static) или виртуальные (virtual) модификаторы

НЕАБСТРАКТНЫЕ МЕТОДЫ АБСТРАКТНОГО КЛАССА

- **Неабстрактные методы абстрактного класса** имеют реализацию и обеспечивают поведение по умолчанию, которое может быть общим для всех классов, наследующих от этого абстрактного класса
  - При необходимости эти методы могут быть переопределены подклассом

ГЕТТЕРЫ И СЕТТЕРЫ В АБСТРАКТНОМ КЛАССЕ

Абстрактные свойства ведут себя также, как и абстрактные методы

НАСЛЕДОВАНИЕ С УЧАСТИЕМ АБСТРАКТНОГО КЛАССА

```
public abstract class Animal
{
    public abstract string MakeSound();
}
```

```
public class Dog : Animal
{
    public override string MakeSound() =>
                           "The dog barks.";
}
```

```
public class Cat : Animal
{
    public override string MakeSound() =>
                           "The cat meows.";
}
```

```
Dog myDog = new Dog();
Console.WriteLine(myDog.MakeSound());

Cat myCat = new Cat();
Console.WriteLine(myCat.MakeSound());
```

КОНСТРУКТОР АБСТРАКТНОГО КЛАССА

```
public abstract class AbstractClass
{
    public string Name { get; set; }

    public AbstractClass(string name) => Name = name;

    public abstract void DoSomething();
}
public class ConcreteClass : AbstractClass
{
    public ConcreteClass(string name) : base(name)
    {
        // Вызов конструктора базового типа.
    }

    // Реализация абстрактного метода.
    public override void DoSomething() =>
        Console.WriteLine("Doing something for " + Name);
}
```

В абстрактном классе могут быть описаны конструкторы и деструкторы

- Не рекомендуется определять открытые (**public**) конструкторы
- Желательно определять внутренние (**internal**) или защищённые (**protected**) конструкторы, чтобы позволить наследникам проводить собственные инициализации и использовать конструктор базового типа только для некоторых наследников

ЗАЧЕМ КОНСТРУКТОРЫ АБСТРАКТНОМУ КЛАССУ

- Нельзя создавать объекты абстрактного класса
  - Зачем он тогда вообще нужен?
- Конструктор абстрактного класса вызывается только в цепочке вызовов конструкторов при создании объектов наследников

```
Cat c = new Cat();
c.Name = "Poor";
```

```
In Animal constructor
In Cat constructor
In cat Name
```

```
public abstract class Animal
{
    protected string _name;
    public Animal() { Console.WriteLine("In Animal constructor"); }
    public abstract string Name { set; }
}
public class Cat: Animal
{
    public Cat() { Console.WriteLine("In Cat constructor"); }
    public override string Name { set { _name = value;
Console.WriteLine("In cat Name"); } }
}
```

ВИРТУАЛЬНЫЕ И АБСТРАКТНЫЕ ЧЛЕНЫ

	Виртуальный член	**Абстрактный член**
Ключевое слово	virtual	abstract
Тело и реализация	Есть тело с реализацией	Нет тела с реализацией – точка с запятой
Переопределение в производном классе	**Может быть** переопределен с использованием модификатора override	**Обязан** быть переопределен с использованием модификатора override
Типы членов	• Методы • Свойства • События • Индексаторы	• Методы • Свойства • События • Индексаторы

ИСПОЛЬЗОВАНИЕ АБСТРАКТНОГО МЕТОДА ДЛЯ ПЕРЕОПРЕДЕЛЕНИЯ В НАСЛЕДНИКАХ

```
1.  // compile with: -target:library
2.  public class D {
3.    public virtual void DoWork(int i) {
4.      // Original implementation.
5.    }
6.  }
7.  public abstract class E : D {
8.    public abstract override void DoWork(int i); }
9.  public class F : E {
10.   public override void DoWork(int i) {
11.     // New implementation.
12.   }
13. }
```

Виртуальный метод из базового класса, класс может быть переопределён в абстрактном классе наследнике помощью абстрактного метода

ПРИМЕР ИЕРАРХИИ

```
public abstract class A
{
    protected int x;
    public abstract void Foo();
}
public abstract class B : A
{
    protected int y;
    public abstract void Bar();
}
```

Здесь возникнет ошибка компиляции, в чем проблема?

```
public class C : B
{
    public void Foo()
    {
        Console.WriteLine(x);
    }
    public void Bar()
    {
        Console.WriteLine(y);
    }
}
```

ПРИМЕР

Наследник абстрактного типа, переопределяет все абстрактные члены

```
public class PolynomFunction: Function
    {
        protected double[] _coefs;
        protected PolynomFunction() { _name = "polynom function"; }
        public PolynomFunction(params double[] coefs)
        {
            _coefs = new double[coefs.Length];
            Array.Copy(coefs,_coefs,coefs.Length);
        }
        public override string Name
        {
            get { return _name; }
        }
        public override double GetFunctionValue(double x)
        {
            double funcValue = _coefs[^1];

            for (int i = 2; i <= _coefs.Length; i++)
            {
                funcValue += _coefs[^i] * x;
                x *= x;
            }
            return funcValue;
        }
    }
```

Базовый класс

```
public abstract class Function
    {
        protected string _name = string.Empty;
        public abstract string Name { get; }
        public abstract double GetFunctionValue(double x);
    }
```

```
PolynomFunction pf = new PolynomFunction(1, 6, 5);
double[] testData = { 0.0, 1, 5, -1, 5 };
foreach(double x in testData)
{
    Console.WriteLine($"f({x}) = {pf.GetFunctionValue(x)}");
}
```

```
f(0) = 5
f(1) = 12
f(5) = 60
f(-1) = 0
f(5) = 60
```

ПОЛИМОРФИЗМ (НАПОМИНАНИЕ)

- Полиморфизм – это концепция теории типов, согласно которой одно и то же имя может обозначать экземпляры разных классов, связанных с общим суперклассом
- Таким образом, любой объект с этим типом может по-разному выполнять общий набор операций
- Используя полиморфизм, одну и ту же операцию можно по-разному реализовывать в классах, образующих иерархию
- В результате подкласс может расширять возможности суперкласса или замещать базовые операции

ПРИМЕР

```
1.  using System;
2.  public abstract class A { public abstract void M1(); }
3.  public class B : A {
4.      public override void M1() { Console.Write("B"); }
5.  }
6.   public class C : A {
7.      public void M1() { Console.Write("C"); }
8.  }
9.  class Program {
10.     static void Main() {
11.         A[] arr = { new B(), new C() };
12.         foreach (A cur in arr) cur.M1();
13.     }
14. }
```

Что будет выведено на экран?

ПРИМЕР, ДОБАВЛЕНИЕ ВТОРОЙ ФУНКЦИИ В ИЕРАРХИЮ

```
public class SinFunction : Function
{
    protected double _a;
    protected double _b;
    protected double _c;
    protected SinFunction() { _name = "trigonometric function"; }
    public SinFunction(params double[] coefs)
    {
        _a = coefs[0];
        _b = coefs[1];
        _c = coefs[2];
    }
    public override string Name
    {
        get { return _name; }
    }
    public override double GetFunctionValue(double x)
    {
        return _a * Math.Sin(_b * x + _c);
    }
}
```

```
PolynomFunction pf = new PolynomFunction(1, 6, 5);
SinFunction sf = new SinFunction(2, 0.5, 1);
double[] testData = { 0.0, 1, 5, -1, 5 };
foreach(double x in testData)
{
    Console.WriteLine($"PolynomicF({x}) = {pf.GetFunctionValue(x)}\t" +
                      $"SinF({x}) = {sf.GetFunctionValue(x):f3}");
}
```

```
PolynomicF(0) = 5       SinF(0) = 1.683
PolynomicF(1) = 12      SinF(1) = 1.995
PolynomicF(5) = 60      SinF(5) = -0.702
PolynomicF(-1) = 0      SinF(-1) = 0.959
PolynomicF(5) = 60      SinF(5) = -0.702
```

РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКИХ ТИПОВ ИСКЛЮЧЕНИЙ

БАЗОВЫЕ КЛАССЫ ДЛЯ ПОЛЬЗОВАТЕЛЬСКИХ ТИПОВ ИСКЛЮЧЕНИЙ

Если ни один из библиотечных типов определения, не удовлетворяет нуждам проектируемой системы, есть возможность реализовать собственный тип, основываясь на System.Exception

- System.Exception
- System.SystemException
- System.ApplicationException // не рекомендуется

- Рекомендуется имена собственных типов исключений снабжать постфиксом Exception:
  - TriangleCanNotBeCreated**Exception**
  - UserListInNullRef**Exception**

КОНСТРУКТОРЫ EXCEPTION

Конструктор	**Описание**
`Exception()`	Инициализирует объект класса Exception
`Exception(String)`	Инициализирует объект класса Exception при помощи сообщения об ошибке
`Exception(SerializationInfo, StreamingContext)`	Инициализирует объект класса Exception на основе сериализованных данных
`Exception(String, Exception)`	Инициализирует объект класса Exception на основе сообщения об ошибке и ссылки на внутреннее исключение, которое стало причиной этого исключения

ПРИМЕР. ИСКЛЮЧЕНИЯ ДЛЯ КЛАССА SEQUENCE (1)

```
public class Sequence
{
    long[] _seqVals = null;
    protected Sequence() { }
    public Sequence(int n)   {
            if (n < 0) { return; }
            _seqVals = new long[n];
            if (n == 1)
            {
                _seqVals[0] = 1;
                return;
            }
            _seqVals[0] = 1;
            _seqVals[1] = -2;
            for (int i = 2; i < n; i++)   {
                _seqVals[i] = 5 * _seqVals[i - 1] + 6 * _seqVals[i - 2];
            }
        }
   public long this[int index]  {  get { return _seqVals[index]; }    }
}
```

- Класс Sequence описывает последовательность: $b_1 = 1, b_2 = -2, \dots, b_n = 5b_{n-1} + 6b_{n-2}$
- Доступ к элементам последовательности организован при помощи индексатора
- Исключение должно появиться в индексаторе для некорректных значений index

ПРИМЕР. ИСКЛЮЧЕНИЯ ДЛЯ КЛАССА SEQUENCE (2)

```
/// <summary>
/// Исключение для обращения по некорректному индексу
/// </summary>
public class SequenceIndexIsOutRangeException : Exception
{
    public SequenceIndexIsOutRangeException() { }
    public SequenceIndexIsOutRangeException(string message) : base(message) { }
    public SequenceIndexIsOutRangeException(string message, Exception exception)
                                                : base(message, exception) { }
}
```

Создаём три конструктора

Модифицируем код индексатора, на некорректных данных выбрасываем исключение

```
public long this[int index]
{
    get
    {
        if (index < 0 || index > _seqVals.Length - 1)
        {
            throw new SequenceIndexIsOutRangeException();

        }
        return _seqVals[index];
    }
}
```

ПРИМЕР. ИСКЛЮЧЕНИЯ ДЛЯ КЛАССА TRIANGLE

```
[Serializable]
public class TriangleException : Exception          // System.ApplicationException
{
    public TriangleException() : base() { }

    public TriangleException(string message) : base(message) { }

    public TriangleException(string message, Exception innerException)
        : base(message, innerException) { }

    public TriangleException(System.Runtime.Serialization.SerializationInfo info,
                             System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }

    public override string ToString() {
        return $"ОШИБКА ВРЕМЕНИ ИСПОЛНЕНИЯ({GetType()}): {Message} в {StackTrace}";
    }
}
```

АЛГОРИТМ СОЗДАНИЯ СОБСТВЕННОГО КЛАССА ИСКЛЮЧЕНИЙ

1. Создать сериализуемый класс, наследник Exception
   1. Атрибут [Serializable]
   2. Идентификатор класса заканчивается постфиксом Exception
2. Добавляем умалчиваемые конструкторы (3 обязательных)
3. Добавляем любые другие свойства и конструкторы класса
4. Как добавить локализованные сообщения в исключениях читаем тут (https://learn.microsoft.com/ru-ru/dotnet/standard/exceptions/how-to-create-localized-exception-messages)

ПРОВЕРЯЕМ СЕБЯ

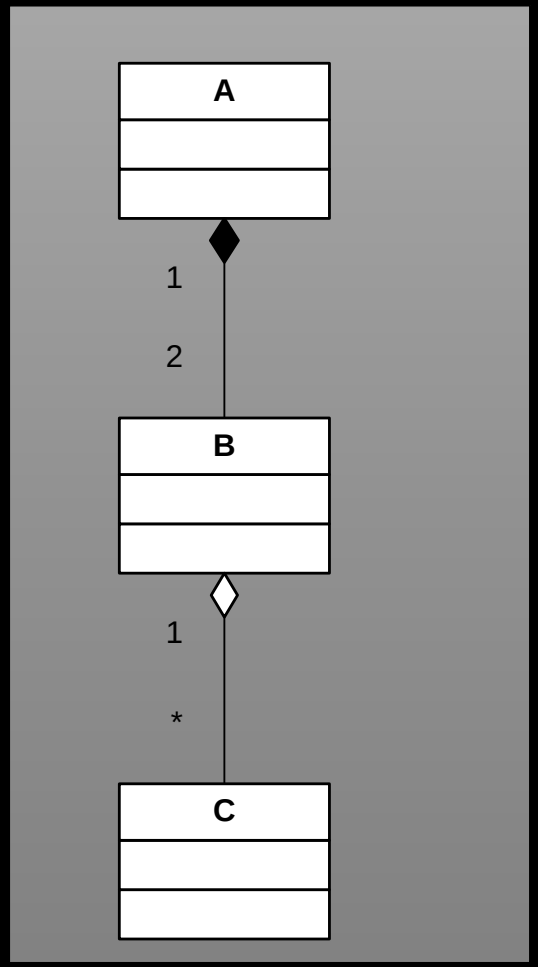
Наследование, агрегация, абстрактный класс

Какие операции со ссылочными переменными в теле метода Main() не приведут к ошибке компиляции?

```
using System;
class A {
    ...
}
class B : A {
    ...
}
class C : B {
    ...
}
public class Program {
    public static void Main() {
        A x1 = new A();
        B x2 = new B();
        C x3 = new C();
        ...
    }
}
```

1. x1 = x2;
2. x1 = x2 = x3;
3. x3 = x2;
4. x3 = x1;
5. x2 = x3;

В каком отношении находятся классы, изображенные на диаграмме?



1. Класс B наследует класс A, а класс C наследует класс C – многоуровневое наследование
2. Классы A и B находятся в отношении композиции, а классы B и C находятся в отношении агрегации
3. Классы A и B находятся в отношении агрегации, а классы B и C находятся в отношении композиции
4. Классы A и B и классы B и C находятся в отношении агрегации
5. Классы A и B и классы B и C находятся в отношении композиции

В результате работы следующего фрагмента программы на экран будет выведено:

```
using System;
class A{
    public int a = 0;
    protected A() { Console.Write(a++); }
}
class B : A {  protected B() { Console.Write(++a); } }
class C : B {  public C() { Console.Write(a--); } }
class Program
{
    static void Main(string[] args)
    {
        C x = new C();
    }
}
```

В результате работы следующего фрагмента программы на экран будет выведено:

```
using System;
class A {
    int a=0;
    protected A() { Console.Write(a++); }
}
class B : A { protected B() {Console.Write(++a); } }
class C : B { public C() { Console.Write(a--); } }
    class Program
    {
        static void Main(string[] args)
        {
            C x = new C();
        }
    }
```

В результате работы следующего фрагмента программы на экран будет выведено:

```
using System;
abstract class A
{
    protected A() { Console.Write("I'm"); }
    public virtual void show() { Console.Write(" here"); }
}
class B : A { public void show() { Console.Write("I am"); } }
class Program
{
    static void Main(string[] args)
    {
        A x;
        B y = new B();
        x = y;
        y.show();
    }
}
```

В результате работы следующего фрагмента программы на экран будет выведено:

```
using System;
class A
{
     protected A() { Console.Write("I'm"); }
     public virtual void show() { Console.Write(" here"); }
}
class B : A { public void show(bool flag) { Console.Write("I am"); } }
class Program
{
     static void Main(string[] args)
     {
          A linkA = new B();
          linkA.show();
     }
}
```

В результате работы следующего фрагмента программы на экран будет выведено:

```
using System;
public abstract class A
{
    protected int a = 1;
    public abstract void Show();
}
public class B : A { public override void Show() { Console.Write(a); } }
public class C : B
{
    int q = 2;
    new public void Show() { Console.Write(q + a);}
}
class Program
{
    static void Main(string[] args)
    {
        B b = new B();
        C c = new C();
        A[] x = { b, c };
        x[0].Show(); b.Show();
        x[1].Show(); c.Show();
    }
}
```

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Материалы лекций Подбельского В.В., Дударева В.А.
- C# abstract class and method (https://www.programiz.com/csharp-programming/abstract-class)
- Материалы с сайта (https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/abstract)
- Лучшие практики (https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions)
- Источник примера (https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members)