Лабораторная работа №4

Цель работы:

Познакомиться с механизмом сериализации классов. Научиться использовать исключения в методах классов.

Необходимые теоретические сведения

GUID

GUID (Globally Unique Identifier) — статистически уникальный 128-битный идентификатор. Его главная особенность — уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов. Хотя уникальность каждого отдельного GUID не гарантируется, общее количество уникальных ключей настолько велико (2128 или 3,4028×1038), что вероятность того, что в мире будут независимо сгенерированы два совпадающих ключа, крайне мала.

«GUID» называют некоторые реализации стандарта, имеющего название Universally Unique Identifier (UUID).

В тексте GUID записывается в виде строки из тридцати двух шестнадцатеричных цифр, разбитой на группы дефисами и опционально окружённой фигурными скобками:

{6F9619FF-8B86-D011-B42D-00CF4FC964FF}

Исключения

В C# ошибки в программе в среде выполнения передаются через программу с помощью механизма, который называется исключениями. Исключения вызываются кодом, который встречает ошибку, и перехватываются кодом, который может ее исправить. Исключения могут вызываться средой выполнения .NET или кодом в программе. Вызванное исключение передается вверх по стеку вызовов, пока не будет найден соответствующий оператор catch. Не перехваченные исключения обрабатываются универсальным обработчиком исключений, предоставляемым системой, которая отображает диалоговое окно.

Исключения представляются классами, производными от Exception. Этот класс определяет тип исключения и содержит свойства с подробными сведениями об исключении. При вызове исключения создается экземпляр производного класса, а также могут настраиваться свойства исключения. После этого с помощью ключевого слова throw вызывается объект.

Если нас не устраивают встроенные типы исключений, то мы можем создать свои типы. Базовым классом для всех исключений является класс Exception, соответственно для создания своих типов мы можем унаследовать данный класс.

Допустим, у нас в программе будет ограничение по возрасту:

Вставить примеры отсюда

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
```

```
            Person p = new Person { Name = "Tom", Age = 17 };
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка: {ex.Message}");
        }
        Console.Read();
    }
}
class Person
{
    private int age;
    public string Name { get; set; }
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
            {
                throw new Exception("Лицам до 18 регистрация запрещена");
            }
            else
            {
                age = value;
            }
        }
    }
}
```

В классе Person при установке возраста происходит проверка, и если возраст меньше 18, то выбрасывается исключение. Класс Exception принимает в конструкторе в качестве параметра строку, которое затем передается в его свойство Message.

Но иногда удобнее использовать свои классы исключений. Например, в какой-то ситуации мы хотим обработать определенным образом только те исключения, которые относятся к классу Person. Для этих целей мы можем сделать специальный класс PersonException:

```
class PersonException : Exception
{
    public PersonException(string message)
        : base(message)
    { }
}
```

По сути класс кроме пустого конструктора ничего не имеет, и то в конструкторе мы просто обращаемся к конструктору базового класса Exception, передавая в него строку message. Но теперь мы можем изменить класс Person, чтобы он выбрасывал исключение именно этого типа и соответственно в основной программе обрабатывать это исключение:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Person p = new Person { Name = "Tom", Age = 17 };
        }
        catch (PersonException ex)
        {
            Console.WriteLine("Ошибка: " + ex.Message);
```

```
        }
        Console.Read();
    }
}
class Person
{
    private int age;
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
                throw new PersonException("Лицам до 18 регистрация запрещена");
            else
                age = value;
        }
    }
}
```

Однако необязательно наследовать свой класс исключений именно от типа Exception, можно взять какой-нибудь другой производный тип. Например, в данном случае мы можем взять тип ArgumentException, который представляет исключение, генерируемое в результате передачи аргументу метода некорректного значения:

```
class PersonException : ArgumentException
{
    public PersonException(string message)
        : base(message)
    { }
}
```

Каждый тип исключений может определять какие-то свои свойства. Например, в данном случае мы можем определить в классе свойство для хранения устанавливаемого значения:

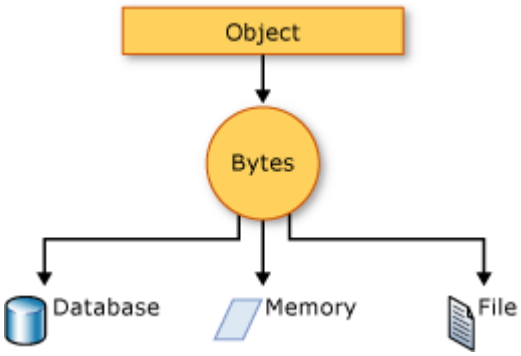
```
class PersonException : ArgumentException
{
    public int Value { get; }
    public PersonException(string message, int val)
        : base(message)
    {
        Value = val;
    }
}
class Person
{
    public string Name { get; set; }
    private int age;
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
                throw new PersonException("Лицам до 18 регистрация запрещена", value);
            else
                age = value;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        try
```

```
        {
            Person p = new Person { Name = "Tom", Age = 13 };
        }
        catch (PersonException ex)
        {
            Console.WriteLine($"Ошибка: {ex.Message}");
            Console.WriteLine($"Некорректное значение: {ex.Value}");
        }
        Console.Read();
    }
}
```

Сериализация

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Эта операция предназначена для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется десериализацией.

На этом рисунке показан общий процесс сериализации.



Объект сериализуется в поток, который служит для передачи данных. Поток также может содержать сведения о типе объекта, в том числе о его версии, языке и региональных параметрах, а также имени сборки. В этом формате потока объект можно сохранить в базе данных, файле или памяти.

Сериализация позволяет разработчику сохранять состояние объекта и воссоздавать его при необходимости. Это полезно для длительного хранения объектов или для обмена данными. Посредством сериализации разработчик может выполнять следующие действия:

- Отправка объекта в удаленное приложение с помощью веб-службы
- Передача объекта из одного домена в другой
- Передача объекта через брандмауэр в виде строки JSON или XML
- Хранение сведений о безопасности и пользователях между приложениями

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable.

При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом NonSerialized.

При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут.

```
[Serializable]
class Worker : Person
```

Формат сериализации

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

- бинарный
- SOAP
- xml
- JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс BinaryFormatter, для формата SOAP - класс SoapFormatter, для xml - XmlSerializer, для json - DataContractJsonSerializer.

Индексаторы

Индексаторы позволяют индексировать экземпляры класса или структуры точно так же, как и массивы. Индексированное значение можно задавать или получать без явного указания типа или экземпляра элемента. Индексаторы действуют как свойства, за исключением того, что их акцессоры принимают параметры.

В следующем примере определяется универсальный класс с простыми акцессорами get и set для назначения и получения значений. Класс Program создает экземпляр этого класса для хранения строк.

```
using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get { return arr[i]; }
        set { arr[i] = value; }
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World";
        Console.WriteLine(stringCollection[0]);
    }
}
```

```
// The example displays the following output:
//       Hello, World.
```

Задание

Написать C# программу на основе лабораторной №1,2,3 реализующую работу с классами по вариантам.

Каждый из классов должен удовлетворять следующим требованиям:

1. Создать интерфейс содержащий атрибут GUID.
2. Абстрактный родительский класс наследовать от созданного интерфейса.
3. В абстрактном классе в конструкторе реализовать метод генерации GUID.
4. Создать класс-хранилище объектов четырёх типов.
5. В классе реализовать индексатор позволяющий получать все объекты указанного типа.
6. Реализовать класс исключения возникающий при добавлении уже существующего экземпляра класса.
7. Реализовать методы сериализации данных класса хранилища в json-файл. И десериализации данных из файла-хранилища.

Разработанная программа должна быть консольной, позволяющей на выбор добавлять объекты классов, при наличии точно такого же экземпляра сообщать об этом пользователю. Кроме того должна быть возможность просмотреть все добавленные объекты заданного класса. При закрытии программы данные о введенных классах не должны теряться. Т.е. при повторном открытии программы ранее введённые данные должны сохраниться.

В отчёт обязательно построить диаграмму классов, описывающую иерархию созданных классов.

Дополнительное задание:

8. В классе реализовать индексатор позволяющий получать объект по указанию GUID типа.
9. Реализовать индексатор в классе позволяющий получать элемент хранилища по его типу и индексу в массиве данных.
10. Реализовать индексатор в классе позволяющий получать элемент хранилища по его типу и GUID

Варианты:

1. Студент, преподаватель, персона, заведующий кафедрой
2. Служащий, персона, рабочий, инженер
3. Рабочий, кадры, инженер, администрация
4. Деталь, механизм, изделие, узел
5. Организация, страховая компания, нефтегазовая компания, завод
6. Журнал, книга, печатное издание, учебник
7. Тест, экзамен, выпускной экзамен, испытание
8. Место, область, город, мегаполис
9. Игрушка, продукт, товар, молочный продукт
10. Квитанция, накладная, документ, счет
11. Автомобиль, поезд, транспортное средство, экспресс
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель
13. Республика, монархия, королевство, государство

14. Млекопитающее, парнокопытное, птица, животное
15. Корабль, пароход, парусник, корвет