GeekBrains

Введение в C#

Работа с файловой подсистемой. Сериализация

На этом уроке

1. Расширим знания о классах и объектах прежде, чем мы приступим к работе с файловой системой и рассмотрим понятие сериализации.
2. Рассмотрим классы и объекты в базовом приближении.
3. Познакомимся с новыми операторами, облегчающими работу с объектами.

Оглавление

Класс и объект

Классы и объекты берут своё начала из ООП. Класс — это набор полей и методов, тесно связанных между собой. Поля класса — это своего рода переменные, объявленные в области видимости класса. Поля, как правило, доступны для чтения и записи только в пределах класса. Для взаимодействия с другими частями приложения класс может иметь свойства. Это обёртки вокруг полей, предоставляющие доступ извне к полям класса для чтения или модификации.

Класс — это своего рода чертёж, по которому в дальнейшем создаются объекты. Чаще всего идеи ООП легко укладываются на нашу картину мира, в том числе классы и объекты. Поэтому рассмотрим, что такое классы и объекты на реальном примере.

Многие термины пришли в программирование из мира строительства и архитектуры зданий. Например, архитектор, фасад, сборка и многие другие. Поэтому представим, как строятся типовые дома.

Сначала архитектор создаёт чертёж дома. В нём указываются свойства будущего жилья: размеры дома, количество этажей и подъездов, наличие лифта и многие другие параметры. Затем, по чертежу возводятся здания. Это может быть один многоэтажный дом или целый микрорайон из строений, построенных по одному и тому же чертежу.

В таком сравнении, класс — это чертёж дома. В нём описаны все свойства: количество этажей, подъездов и прочие параметры. Применяются методы: открыть дверь, вызвать лифт, включить свет и так далее. Описываются внутренние поля, скрытые от пользователей дома — параметры, которые не предусматривают их изменение извне: мастер-пароль от дверей подъезда и так далее.

Создадим примитивный класс, описывающий будущий жилой дом.

Важно!

1. *Кроме кода, пример содержит комментарии документации, подробнее о которых вы можете узнать по ссылке в дополнительных источниках к уроку.*
2. *Далее нам будет встречаться ключевое слово public. Оно означает, что класс, метод или свойство будут доступны за пределами класса.*

```
using System;
namespace HelloWorld
{
    /// <summary>
    /// Здание
    /// </summary>
    public class Building
    {
        /// <summary>
        /// Количество этажей
        /// </summary>
```

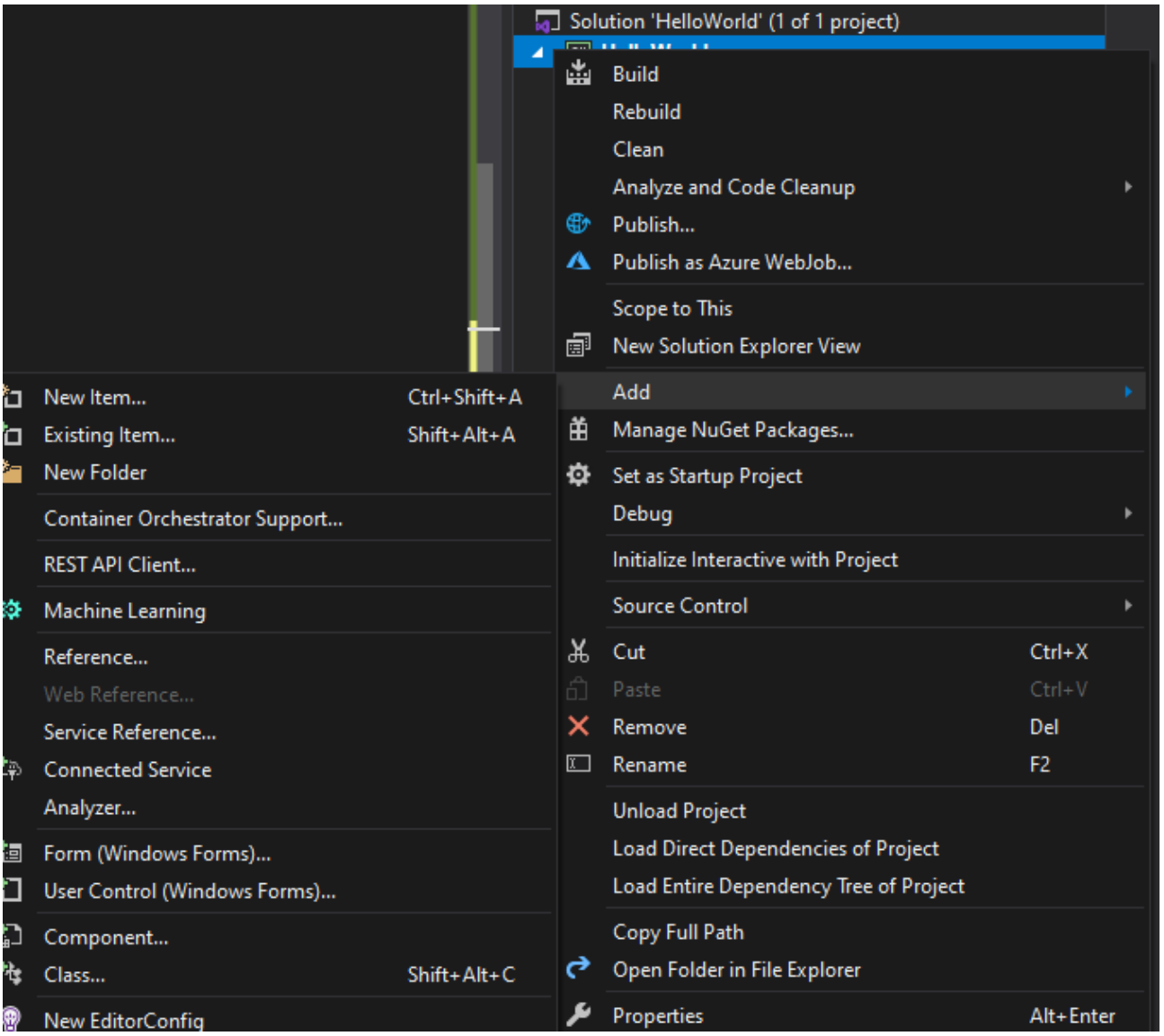
```
        public int Floors { get; }

        /// <summary>
        /// Количество подъездов
        /// </summary>
        public int Entrances { get; }

        /// <summary>
        /// Признак отапливаемости здания
        /// </summary>
        public bool IsHeatable { get; set; }

        /// <summary>
        /// Адрес здания
        /// </summary>
        public string Address { get; set; }

        /// <summary>
        /// Создает здание с заданными параметрами
        /// </summary>
        /// <param name="floors">Количество этажей</param>
        /// <param name="entrances">Количество подъездов</param>
        public Building(int floors, int entrances)
        {
            Floors = floors;
            Entrances = entrances;
        }

        /// <summary>
        /// Вызывает лифт на заданный этаж
        /// </summary>
        /// <param name="entrance">Номер подъезда</param>
        /// <param name="floor">Номер этажа</param>
        public void GetElevator(int entrance, int floor)
        {
            Console.WriteLine("3..2..1..Ding!");
        }

        /// <summary>
        /// Открывает дверь заданного подъезда
        /// </summary>
        /// <param name="entrance">Номер подъезда</param>
        public void OpenDoor(int entrance)
        {
            Console.WriteLine("Please come in!");
        }
    }
}
```

Здесь стоит обратить внимание на несколько вещей. `Floors`, `Entrances`, `IsHeatable`, `Address` — это свойства будущего дома, а `GetElevator` и `OpenDoor` — его методы. Кроме свойств и методов, мы описали ещё один метод, имеющий особую сигнатуру:

```
public Building(int floors, int entrances)
{
    Floors = floors;
    Entrances = entrances;
}
```

Такой метод называется конструктором. Имя конструктора должно совпадать с именем класса, а в качестве аргументов могут быть указаны обязательные аргументы. В результате вызова конструктора создаётся экземпляр класса — объект.

Рассмотрим, как происходит создание объектов и разберём все ключевые моменты. Для начала создадим в нашем проекте новый класс. Кликнем правой кнопкой мыши по проекту и выберем Add -> Class. В качестве альтернативы можно выбрать Add -> New Item… и далее выбрать Class:



Укажем имя класса `Building` и нажмём Add. Затем добавим в него содержимое, описанное выше.

Теперь перейдём в метод `Main` и опишем в нём следующий код:

```
static void Main()
{
    Building house = new Building(9, 3);
    house.Address = "ул. Ленина, 12";
    house.OpenDoor(1);
    house.IsHeatable = true;
}
```

В первой строке мы создаём объект типа `Building` с помощью оператора new и вызова конструктора класса `Building`. В него мы передаём значения для ранее объявленных аргументов.

Оператор `new` «за кулисами» выполняет выделение памяти под будущий объект и все объявленные в нём поля и свойства. Затем вызывает конструктор, который устанавливает значения для свойств `Floors` и `Entrances`. Обратите внимание, как эти свойства объявлены в коде:

```
public int Floors { get; }
public int Entrances { get; }
```

После объявления свойства в фигурных скобках указываются доступные методы доступа. `get` означает, что значение свойства может быть прочитано. Ключевое слово `set` указывает, что значение свойства может быть записано. У свойств `Floors` и `Entrances` указано только ключевое слово `get`, так как мы не можем динамически изменять количество этажей и подъездов у готового здания. Напротив, у готового здания может измениться адрес (переименование улицы). А отапливаемость здания — сезонный момент, поэтому эти свойства доступны как для чтения, так и для записи:

```
public bool IsHeatable { get; set; }
public string Address { get; set; }
```

В `Main` после «постройки» здания мы можем вызывать методы и задавать значения свойств, доступных для записи:

```
house.IsHeatable = true;
Console.WriteLine(house.Entrances); // 3
house.Entrances = 4; // ошибка компиляции, т.к. Entrances только для чтения
```

Теперь, когда мы знаем базовую информацию о классах и объектах, то можем перейти к работе с файлами.

Чтение и запись файлов

Возможность чтения и записи файлов (текстовых и двоичных) предоставляет класс `File` из пространства имён `System.IO`. Перед выполнением следующих примеров нужно убедиться, что в классе Program подключено это пространство имён:

```
using System.IO;
```

Чтение и запись текстовых файлов

Класс `File` предоставляет следующие методы для чтения и записи текстовых файлов:

- `ReadAllText` — открывает текстовый файл, считывает весь текст файла в виде строки.
- `ReadAllLines` — открывает текстовый файл, считывает весь текст файла в виде набора строк.
- `WriteAllText` — создаёт новый файл и записывает в него указанную строку. Если файл уже имеется, он будет перезаписан.
- `WriteAllLines` — создаёт новый файл и записывает в него указанный массив строк. Если файл уже есть, он будет перезаписан. Каждый элемент массива записывается с новой строки.
- `AppendAllText` — добавляет текст в указанный файл. Если файла нет, он будет создан.
- `AppendAllLines` — добавляет строки в указанный файл. Если файл отсутствует, он будет создан. Каждый элемент массива записывается с новой строки.

Пример работы с методами:

```
static void Main(string[] args)
{
    string filename = "text.txt";
    File.WriteAllText(filename, "str1"); // записываем в файл строку

       File.AppendAllText(filename, Environment.NewLine); // вставляем перенос
строки
     File.AppendAllLines(filename, new[] { "str2", "str3" }); // добавляем еще
две строки

    string fileText = File.ReadAllText(filename);

    Console.WriteLine(fileText);

    string[] fileLines = File.ReadAllLines(filename);

    Console.WriteLine(fileLines[1]); // str2
}
```

Текстовый файл будет создан рядом с исполняемым файлом приложения.

Чтение и запись двоичных файлов

Класс `System.IO.File` также предоставляет методы для чтения и записи двоичных файлов — `File.ReadAllBytes` и `File.WriteAllBytes`:

```
byte[] array = { 1, 2, 3, 5, 7, 9, 11 };
File.WriteAllBytes("bytes.bin", array);
```

```
byte[] fromFile = File.ReadAllBytes("bytes.bin"); // { 1, 2, 3, 5, 7, 9, 11 }
```

Форматы хранения данных

Ранее мы рассмотрели возможности языка C# по чтению и записи текстовых и двоичных файлов. Это были простые операции чтения и записи однородной информации — строк или набора байт. На практике зачастую возникает необходимость хранить в файлах структурированный набор информации для упрощения их обработки и обмена с другими программами.

Процесс сохранения структурированной информации в файле называется сериализацией. Восстановление структурной информации из файла — десериализация. Сериализация имеет широкое применение при передаче данных по сети, а также для хранения настроек приложения.

Сохранение может производиться как в текстовом формате, так и в бинарном. Далее мы рассмотрим текстовые форматы сериализации, а также затронем её бинарную версию. Последняя будет изучена более детально в других курсах.

Наиболее распространённые текстовые форматы хранения и передачи данных — `XML` и `JSON`. Оба этих формата поддерживают хранение сложных вложенных структур данных. Они имеют поддержку базовых типов данных. Эти форматы легко воспринимаются людьми благодаря простой структуре.

`JSON` — текстовый формат обмена данными, основанный на JavaScript. Однако формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате `JSON`. Благодаря своей лаконичности по сравнению с `XML` формат `JSON` может быть более подходящим для сериализации сложных структур.

Пример `JSON`-файла:

```
{
    "name": "Jack",
    "surname": "Sparrow",
    "title": "Captain"
}
```

`XML` — это текстовый формат обмена данными, составленный на одноимённом языке разметки. Структура файла и его параметры прописываются с помощью тегов, атрибутов и препроцессоров. Напоминает `HTML`-разметку.

Пример `XML`-файла:

```
<Role>
    <Name>Jack</Name>
```

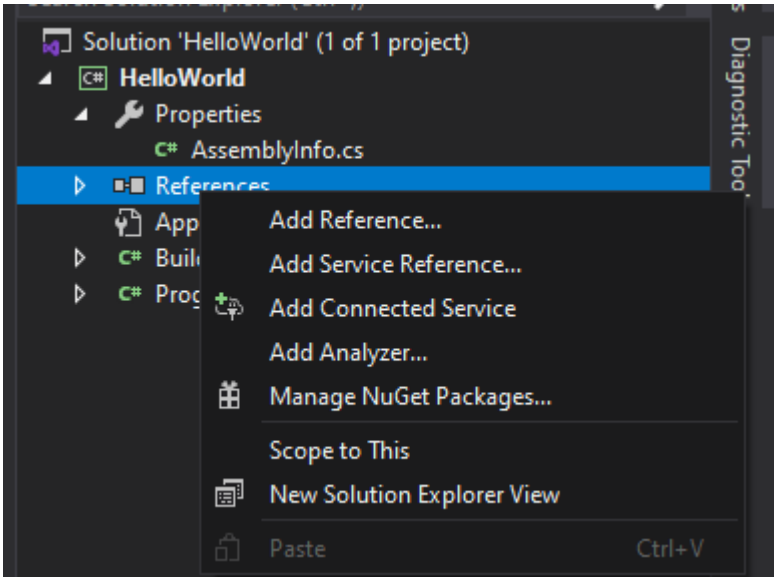
```
    <Surname>Sparrow</Surname>
    <Title>Captain</Title>
</Role>
```

В отличие от `JSON`, корректный `XML` требует наличия корневого элемента, в этом примере — `Role`.

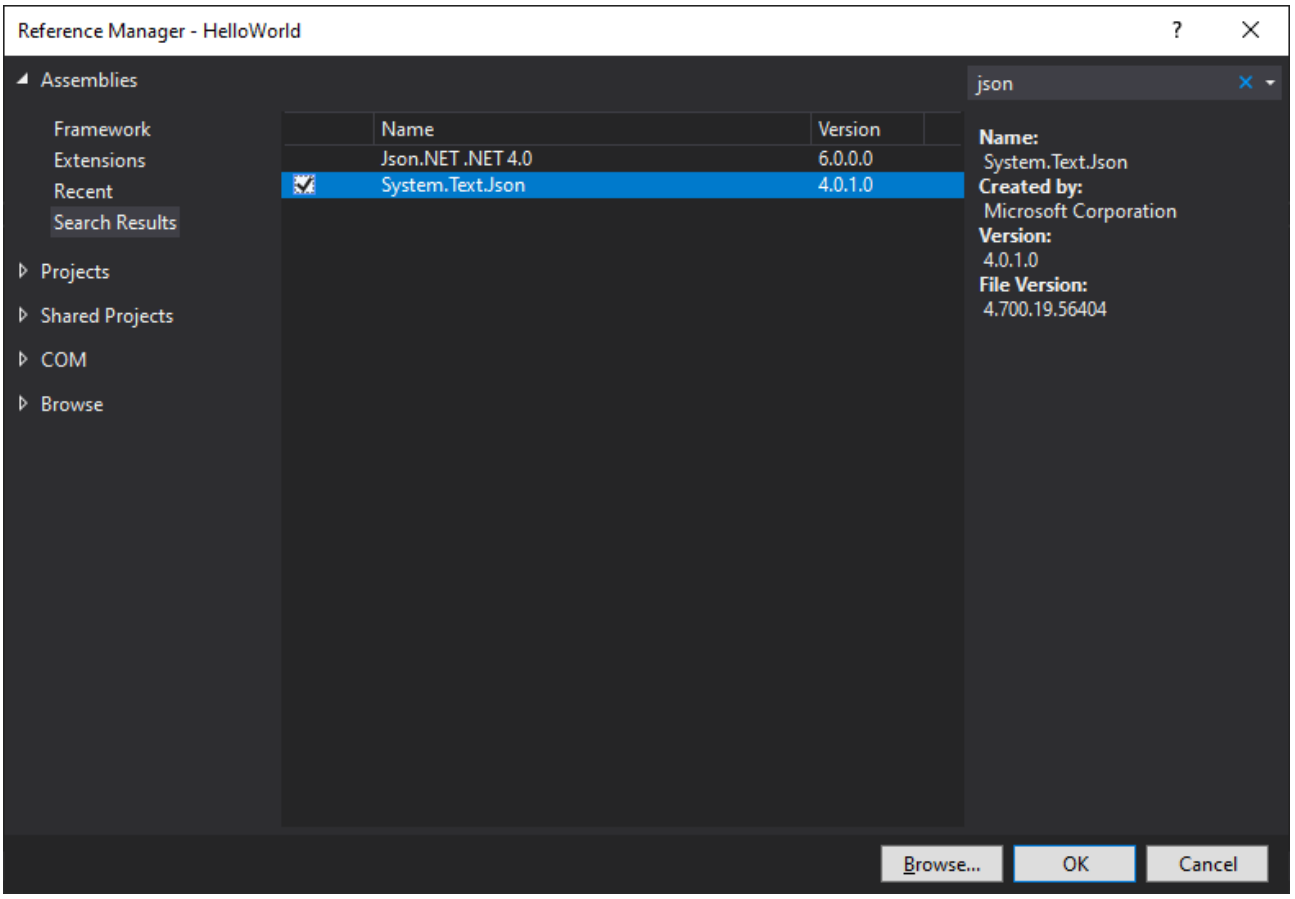
Сериализация и десериализация в .NET

JSON-сериализация

Для сериализации в формат `JSON` в .NET существует класс `JsonSerializer` из пространства имён `System.Text.Json`. По умолчанию сборка, содержащая это пространство имён, не добавляется в проект консольного приложения. Поэтому нужно подключить сборку вручную. Для этого щёлкните правой кнопкой мыши по пункту References в структуре проекта в окне Solution Explorer и выберите Add Reference:



В открывшемся окне наберите json в поисковое поле и выберите сборку `System.Text.Json`:



Аналогичным образом найдите и выберите сборку `System.Memory`. Если найдётся несколько сборок, выберите любую. Эта сборка может потребоваться при десериализации.

Закройте окно кнопкой OK. Затем добавьте следующие пространства имён в файл `Program.cs`:

```
using System.IO;
using System.Text.Json;
```

Ранее мы описывали класс `Building`. Давайте сериализуем объект этого класса:

```
using System.IO;
using System.Text.Json;
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            Building house = new Building(9, 3);
            house.Address = "ул. Ленина, 12";
            string json = JsonSerializer.Serialize(house);
            File.WriteAllText("house.json", json);
        }
    }
}
```

Посмотрим на результат:

```
{
    "Floors": 9,
    "Entrances": 3,
    "IsHeatable": false,
    "Address": "\u0443\u043B. \u041B\u0435\u043D\u0438\u043D\u0430, 12"
}
```

Кириллические символы в адресе заменились на коды Unicode. Это важно для их корректного отображения без привязки к кодировке текстового файла. Попробуем десериализовать файл:

```
string json = File.ReadAllText("house.json");
Building house = JsonSerializer.Deserialize<Building>(json);
```

При запуске произойдёт исключительная ситуация:

`Unhandled Exception: System.NotSupportedException: Deserialization of reference types without parameterless constructor is not supported.`

Дело в том, что `JsonSerializer` пытается создать объект `Building`, который имеет только параметризованный конструктор (конструктор с аргументами). При десериализации сначала создаётся объект путём вызова конструктора по умолчанию. Затем в свойства этого объекта заносятся значения из `JSON`-файла. Но чтобы избежать ошибок, требуется явное описание конструктора по умолчанию (конструктор без параметров). Поэтому добавим этот конструктор в класс `Building`, например, перед имеющимся конструктором:

```
...
public string Address { get; set; }
...
public Building()
{
    Floors = 3;
    Entrances = 1;
}
```

В теле конструктора мы задаём произвольные значения для этажности и количества подъездов. Попробуем десериализовать файл ещё раз и добавить для наглядности логирование в консоль:

```
string json = File.ReadAllText("house.json");
Building house = JsonSerializer.Deserialize<Building>(json);
System.Console.WriteLine(house.Address); // ул. Ленина, 12
System.Console.WriteLine(house.Floors); // 3
System.Console.WriteLine(house.Entrances); // 1
```

Обратите внимание, что значения для `Floors` и `Entrances` не соответствуют указанным значениям в `JSON`-файле. Дело в том, что мы объявили эти свойства доступными только для чтения. Их значения можно задать только в конструкторе. Поэтому `JsonSerializer` не смог задать им соответствующие значения. Обновим объявление этих свойств в классе `Building`, добавив сеттер (ключевое слово set):

```
...
public int Floors { get; set; }
...
public int Entrances { get; set; }
```

Запустим наш код ещё раз. Теперь значения этих свойств совпадают со значениями, указанными в `JSON`-файле.

`XML`-сериализация

Добавим данные пространства имён в `Program.cs`:

```
using System.Xml.Serialization;
using System.IO;
```

Процедура сериализации в формате XML немного отличается от JSON-сериализации. `XmlSerializer` не возвращает строку напрямую. Он использует определённый буфер для хранения результата, в качестве которого будем использовать `StringWriter`. Не будем подробно

останавливаться на этом классе, так как потоковый ввод-вывод выходит за рамки текущего урока. Всё, что необходимо знать об этом классе сейчас:

- он хранит большие объёмы текстовых данных в памяти;
- в дальнейшем позволяет сохранить их в файл или в заданную переменную.

Следующий пример сохраняет объект house в файл `house.xml`:

```
Building house = new Building(12, 4);
house.Address = "ул. Мира, 24";
house.IsHeatable = true;

StringWriter stringWriter = new StringWriter();
XmlSerializer  serializer = new XmlSerializer(typeof(Building));
serializer.Serialize(stringWriter, house);
string xml = stringWriter.ToString();
File.WriteAllText("house.xml", xml);
```

В результате получится такой `XML`-документ, вначале указаны служебные сведения:

```
<?xml version="1.0" encoding="utf-16"?>
<Building                              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Floors>12</Floors>
  <Entrances>4</Entrances>
  <IsHeatable>true</IsHeatable>
  <Address>ул. Мира, 24</Address>
</Building>
```

Попробуем прочитать этот документ:

```
string xmlText = File.ReadAllText("house.xml");
StringReader stringReader = new StringReader(xmlText);
XmlSerializer serializer = new XmlSerializer(typeof(Building));
Building house = (Building)serializer.Deserialize(stringReader);
System.Console.WriteLine(house.Address); // ул. Мира, 24
System.Console.WriteLine(house.Floors); // 12
System.Console.WriteLine(house.Entrances); // 4
```

Метод `Deserialize` возвращает стандартный `Object`. Поэтому мы приводим его к типу `Building`. Так как в разделе `JSON`-сериализации мы добавили конструктор по умолчанию и разрешили записывать значения в свойства класса `Building`, `XML`-десериализация проходит успешно.

Бинарная сериализация

Добавим данные пространства имен в `Program.cs`:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

Чтобы класс `Building` можно было сериализовать в бинарный формат, нужно добавить ему атрибут `Serializable` из пространства имён `System`:

```
using System;
namespace HelloWorld
{
    /// <summary>
    /// Здание
    /// </summary>
    [Serializable]
    public class Building
...
```

Для бинарной сериализации используется класс `BinaryFormatter`. Как и в случае с `XML`-сериализацией, бинарная сериализация реализована на файловых потоках:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            Building house = new Building(5, 2);
            house.Address = "ул. Победы, 14";
            BinaryFormatter formatter = new BinaryFormatter();
                                formatter.Serialize(new  FileStream("house.bin",
FileMode.OpenOrCreate), house);
        }
    }
}
```

Для десериализации используется метод `Deserialize`:

```
BinaryFormatter formatter = new BinaryFormatter();
Building building = (Building)formatter.Deserialize(new FileStream("house.bin",
FileMode.Open));
Console.WriteLine(building.Address); // ул. Победы, 14
```

Null-условные операторы

Null-условные операторы используются для безопасного доступа к свойствам, которые могут иметь значение null. Использование операторов помогает избежать некоторых исключительных ситуаций во время выполнения программы. Представим, что мы хотим вывести номер дома. При условии, что адрес всегда содержит только название улицы и номер дома, мы можем реализовать это следующим образом:

```
Building house = new Building(5, 2);
house.Address = "Горького, 34";
Console.WriteLine(house.Address.Split(' ')[1]); // 34
```

Здесь мы разделяем строку с адресом на массив строк. Разделителем служит символ пробела. Нулевой элемент будет содержать название улицы и запятую, а первый элемент — номер дома.

Но что, если у дома не указан адрес?

```
Building house = new Building(5, 2);
Console.WriteLine(house.Address.Split(' ')[1]); // Unhandled Exception
```

В таком случае мы попытаемся вызвать несуществующий метод Split, так как house.Address будет равняться null. Нам придётся проверять наличие адреса у дома:

```
Building house = new Building(5, 2);
if (house.Address != null)
{
    Console.WriteLine(house.Address.Split(' ')[1]);
}
```

Теперь наш код не будет завершаться с ошибкой. Представим, что мы не создаём объект house, а принимаем его откуда-нибудь ещё. Например, в результате десериализации или из сети. В этом случае мы должны проверить, что house действительно объект, а не null:

```
static void Main()
{
    PrintHoseNumber(null);

}

static void PrintHoseNumber(Building house)
```

```
{
    if (house == null)
    {
        return;
    }
    if (house.Address == null)
    {
        return;
    }
    Console.WriteLine(house.Address.Split(' ')[1]);
}
```

Код становится более громоздким. Чтобы упростить его, можно применить оператор условного доступа к свойствам и методам объекта. Например, `null`-conditional operator или просто `null`-оператор. Он состоит из знака вопроса и точки — «`?.`». Перепишем наш метод с использованием этого оператора:

```
static void Main()
{
    PrintHouseNumber(null);
    PrintHouseNumber(new Building());
    PrintHouseNumber(new Building { Address = "Горького, 34" });
}

static void PrintHouseNumber(Building house)
{
    Console.WriteLine(house?.Address?.Split(' ')[1]);
}
```

Оператор условного доступа возвращает требуемое значение либо `null`. Через `null`-оператор мы можем безопасно запрашивать вложенные свойства. Он продолжит выполнение оставшейся части выражения только в том случае, если вычислит значение, отличное от `null`.

Null-оператор может применяться к индексаторам массива:

```
Person first = people?[0];
```

В случаях, когда необходимо вернуть значение по умолчанию для какого-либо значения, используется оператор объединения с `null` (`null`-coalescing operator) — «`??`». Оператор возвращает первый операнд, если он отличается от `null`. В противном случае возвращается второй операнд:

```
Building house = new Building();
Console.WriteLine(house.Address ?? "Нет адреса"); // Нет адреса
house.Address = "Тургенева, 18";
Console.WriteLine(house.Address ?? "Нет адреса"); // Тургенева, 18
```

Операции с файлами и директориями

.NET содержит класс `System.IO.Directory`, позволяющий производить различные операции над директориями: создавать, перемещать, получать список содержимого директории. Ранее мы использовали класс `System.IO.File` для чтения и записи файлов. Но этот класс также позволяет осуществлять такие файловые операции, как копирование, перемещение, удаление и прочие, которые мы рассмотрим в этом разделе. Так как большинство операций однотипные, рассмотрим их с помощью комплексного примера, демонстрирующего работу сразу с несколькими методами. Далее будут представлены часто используемые методы класса `Path`, необходимые для операций над строками, содержащими сведения о пути к файлу или каталогу.

Перед началом мы создадим рабочую директорию для наших будущих примеров. Все последующие операции, изменяющие файловую систему, будем производить внутри директории. В последующих примерах будет использоваться директория «`D:\ExampleDir`».

```
string workDir = @"D:\ExampleDir";

Console.WriteLine(Directory.Exists(workDir));   //   Проверяет,   существует   ли
заданная директория

string notesDir = Path.Combine(workDir, "Notes"); // "D:\ExampleDir\Notes"

Directory.CreateDirectory(notesDir); // Создаем вложенную директорию

string noteText = "Этот файл создан автоматически";

string     notePath     =     Path.Combine(notesDir,     "Note     1.txt");     //
"D:\ExampleDir\Notes\Note 1.txt"

File.WriteAllText(notePath, noteText);

string  copyOfNotePath  =  Path.Combine(workDir,  "Copy  of  Note  1.txt");  //
"D:\ExampleDir\Copy of Note 1.txt"
File.Copy(notePath,   copyOfNotePath);   //   Копируем   созданную   заметку   в
"D:\ExampleDir\Copy of Note 1.txt"

Console.WriteLine(File.Exists(copyOfNotePath));  //  Проверяет,  существует  ли
заданный файл

File.Move(copyOfNotePath, Path.Combine(notesDir, "Note 2.txt")); // Перемещаем
заметку в "D:\ExampleDir\Notes\Note 2.txt"
```

```
// Создаем директорию "D:\ExampleDir\Documents" и перемещаем в нее директорию Notes
Directory.CreateDirectory(Path.Combine(workDir, "Documents"));
Directory.Move(notesDir, Path.Combine(workDir, "Documents", "Notes"));

// Перечень всех файлов и папок, вложенных в workDir
string[] entries = Directory.GetFileSystemEntries(workDir, "*", SearchOption.AllDirectories);

for (int i = 0; i < entries.Length; i++)
{
    Console.WriteLine(entries[i]);
}
```

Методы классов `Directory` и `Path` позволяют получать необходимые фрагменты из пути к файлу или каталогу:

```
string note = @"D:\ExampleDir\Documents\Notes\Note 1.txt";

Console.WriteLine(Path.GetFileName(note)); // Note 1.txt
Console.WriteLine(Path.GetFileNameWithoutExtension(note)); // Note 1
Console.WriteLine(Path.GetExtension(note)); // .txt
Console.WriteLine(Directory.GetParent(note));  // D:\ExampleDir\Documents\Notes
```

Создание и распаковка архивов

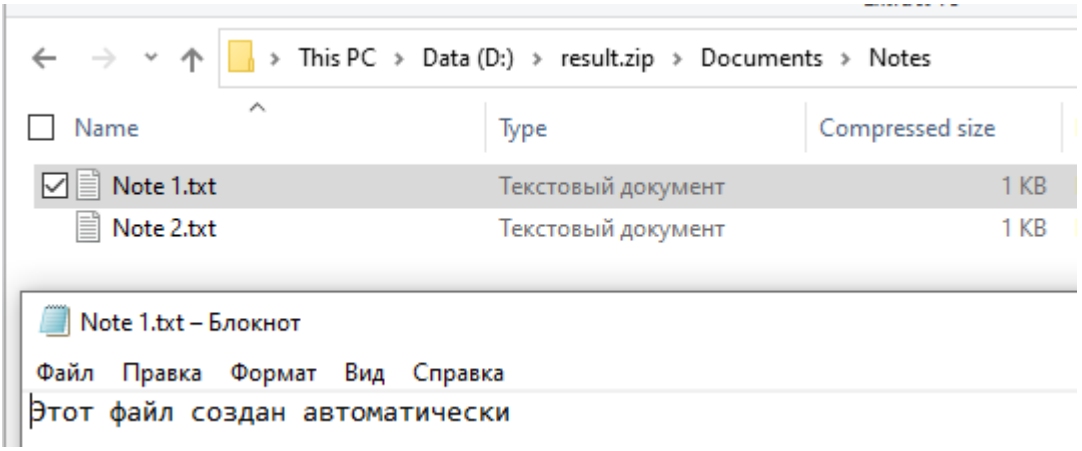
.NET содержит встроенные возможности по созданию и извлечению Zip-архивов. Вначале нам нужно добавить ссылку на сборку `System.IO.Compression.FileSystem`. Мы делали это со сборкой `System.Text.Json` в разделе о сериализации. Далее нужно добавить следующее пространство имён в файл `Program.cs`:

```
using System.IO.Compression;
```

Создадим архив `D:\result.zip` из содержимого каталога «`D:\ExampleDir`»:

```
using System.IO.Compression;
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            ZipFile.CreateFromDirectory(@"D:\ExampleDir", @"D:\result.zip");
        }
    }
}
```

Убедимся, что в архиве и исходной директории находятся одинаковые данные:



Далее распакуем архив в указанную директорию. Если пути нет, он будет создан:

```
using System.IO.Compression;

namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            ZipFile.ExtractToDirectory(@"D:\result.zip", @"D:\Example\Dir");
        }
    }
}
```

Практическое задание

1. Ввести с клавиатуры произвольный набор данных и сохранить его в текстовый файл.
2. Написать программу, которая при старте дописывает текущее время в файл «startup.txt».
3. Ввести с клавиатуры произвольный набор чисел (0...255) и записать их в бинарный файл.
4. (*) Сохранить дерево каталогов и файлов по заданному пути в текстовый файл — с рекурсией и без.
5. (*) Список задач (ToDo-list):
    - написать приложение для ввода списка задач;
    - задачу описать классом `ToDo` с полями `Title` и `IsDone`;
    - на старте, если есть файл `tasks.json/xml/bin` (выбрать формат), загрузить из него массив имеющихся задач и вывести их на экран;
    - если задача выполнена, вывести перед её названием строку «[x]»;
    - вывести порядковый номер для каждой задачи;
    - при вводе пользователем порядкового номера задачи отметить задачу с этим порядковым номером как выполненную;
    - записать актуальный массив задач в файл `tasks.json/xml/bin`.

Используемые источники

1. Documentation comments - C# language specification.
2. Как сериализировать и десериализировать JSON.
3. XmlSerializer.Serialize Метод (System.Xml.Serialization).
4. BinaryFormatter Класс (System.Runtime.Serialization.Formatters.Binary).

Обратная связь

Нам очень важно, чтобы обучение приносило результаты и было комфортным для наших студентов.

Пожалуйста, заполните эту форму, чтобы мы могли сделать наше обучение лучше.

Вы можете оставить обратную связь анонимно, но будет гораздо полезней, если вы укажете ваше имя и адрес электронной почты, чтобы мы могли с вами связаться в случае необходимости.

Ваши персональные данные и результаты опроса будут использованы только авторами курса, методистом и преподавателем для улучшения программы.