Объектно ориентированное программирование на C#

Тема 29. Урок 1. Теория. Работа с CSV файлами.

Введение.
CSV (Comma Separated Values, значения, разделённые запятыми) - это текстовый формат файла, который содержит табличные данные. Он состоит из строк и столбцов, где каждая строка представляет собой набор данных, а каждый столбец - определённое поле. Значения полей разделены запятыми, а в качестве разделителя полей используется символ запятой.

CSV-файлы обычно используются для передачи табличных данных между разными программами или платформами, так как они легки в чтении и редактировании. Они также подходят для импорта и экспорта данных из баз данных и других программ, которые поддерживают работу с табличными данными. Кроме того, CSV-файлы можно открывать и редактировать в текстовых редакторах, что делает их удобными для ручной обработки данных.

Пример содержимого CSV файла.
Вот пример простого CSV-файла с данными о клиентах:

"Имя","Фамилия","Email","Номер телефона"
"Иван","Петров","john.doe@example.com","+1 (123) 456-7890"
"Сергей","Иванов","jane.doe@example.org","+1 (234) 567-8901"
"Елена","Сидорова","joe.bloggs@example.net","+1 (345) 678-9012"

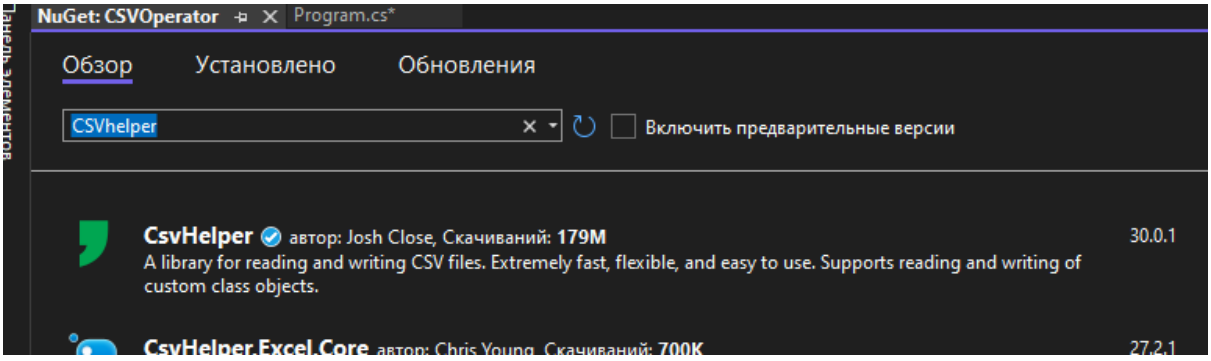
В этом файле, символом-разделителем является запятая.
Заметьте, что текстовые поля (такие как “First Name” и “Last Name”) заключены в двойные кавычки, чтобы предотвратить ошибки, связанные с наличием запятых внутри значений полей.
Здесь, так же как и в Excel таблицах все разделено на строки и столбцы.

К примеру, колонка:
"Иван"
"Сергей"
"Елена"
Является значениями столбца “Имя”.

"Петров"
"Иванов"
"Сидорова"
Является значениями столбца "Фамилия".
И так далее.
То есть, в простом понимании, мы имеем **простую двумерную таблицу с данными и заголовками столбцов**.

Библиотека C# для работы с CSV файлами.
Для работы с CSV-файлами в C# можно использовать библиотеку под названием “CsvHelper”. Она позволяет читать и записывать CSV-файлы, обрабатывать их содержимое и манипулировать данными.

Для работы с библиотекой, необходимо добавить ее с помощью пакетного менеджера в ваш проект:



Определение класса данных.
Создайте класс данных, который будет представлять объекты, которые вы хотите читать или записывать в CSV файл.
Например:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Определение класса данных (Data Class) в программировании часто используется для структурирования и представления данных, а также для упрощения работы с данными в коде.

Запись данных в CSV файл.

Для записи данных в CSV файл, используйте CsvWriter:

```
static void Main(string[] args)
{
    // Создание StreamWriter для записи в файл "output.csv".
    StreamWriter writer = new StreamWriter("output.csv");

    // Создание CsvWriter с настройками и привязка к StreamWriter.
    CsvWriter csvWrite = new CsvWriter(writer, new
CsvConfiguration(CultureInfo.InvariantCulture));

    // Создание списка объектов Person с данными.
    List<Person> record = new List<Person>()
    {
        new Person { Name = "John", Age = 30 },
        new Person { Name = "Alice", Age = 25 }
    };

    // Запись данных из списка record в файл "output.csv" с
использованием CsvWriter.
    csvWrite.WriteRecords(record);

    // Закрытие StreamWriter и файла "output.csv".
    writer.Close();
}
```

Теперь давайте рассмотрим каждую строку кода более подробно:

StreamWriter writer = new StreamWriter("output.csv");

Создание экземпляра StreamWriter, который будет использоваться для записи данных в файл "output.csv".

CsvWriter csvWrite = new CsvWriter(writer, new CsvConfiguration(CultureInfo.InvariantCulture));

Создание экземпляра CsvWriter, который будет использоваться для записи данных в формате CSV. CsvWriter связывается с StreamWriter, который определен в предыдущей строке, чтобы направлять вывод в файл "output.csv".

List<Person> record = new List<Person>() { ... }
Создание списка объектов Person, который представляет данные для записи в CSV файл. В данном случае, список содержит два объекта Person с именем "John" и "Alice", а также их возрастом.

csvWrite.WriteRecords(record);
Запись данных из списка record в файл "output.csv" с использованием CsvWriter. Эта операция **пройдет по каждому элементу в списке и запишет их в файл в формате CSV.**

writer.Close();
Закрытие StreamWriter и файла "output.csv". Данная конструкция позволяет нам закрыть StreamWriter вручную.

Этот код создаст CSV файл "output.csv" с данными из списка объектов Person.

Что касается **new CsvConfiguration(CultureInfo.InvariantCulture)** - это конфигурация, которая настраивает CsvWriter на использование инвариантной культуры (CultureInfo.InvariantCulture) при записи данных в CSV формат.

Когда вы записываете числовые значения (например, числа с плавающей точкой) в CSV файл, формат чисел может различаться в зависимости от текущей культуры вашей операционной системы. Например, в некоторых культурах **вместо точки в качестве разделителя десятичных знаков используется запятая.**

Использование CultureInfo.InvariantCulture гарантирует, что числовые значения будут записаны в формате, не зависящем от текущей культуры операционной системы.
Это важно, если вы планируете, например, читать данные из CSV файла на одной машине и записывать их на другой, и оба компьютера могут использовать разные культуры.

В общем случае, использование CultureInfo.InvariantCulture рекомендуется при работе с CSV файлами, чтобы гарантировать, что числовые значения будут записаны и считаны в ожидаемом формате, независимо от культурных различий.

Таким образом, по итогу, в папке …\SandBox\bin\Debug\net6.0, вашего проекта создается файл output.csv и вот его содержимое:

```
Name,Age
John,30
Alice,25
```

Чтение данных из CSV файла.

Для чтения данных из CSV файла, используйте CsvReader:

```
static void Main(string[] args)
{
    // Создание StreamReader для чтения из файла "output.csv".
    StreamReader reader = new StreamReader("output.csv");

    // Создание CsvReader с настройками и привязка к созданному
StreamReader.
    CsvReader csvRead = new CsvReader(reader, new
CsvConfiguration(CultureInfo.InvariantCulture));

    // Чтение данных из файла "output.csv" и преобразование их в список
объектов Person.
    List<Person> records = csvRead.GetRecords<Person>().ToList();

    // Не забываем закрыть поток файла после работы с ним
    reader.Close();

    // Отображение данных из списка на консоль.
    foreach (Person person in records)
    {
        Console.WriteLine($"Имя: {person.Name}\nВозраст:
{person.Age}");
    }
}
```

Теперь давайте рассмотрим каждую строку кода более подробно:

StreamReader reader = new StreamReader("output.csv");
Создание экземпляра StreamReader, который будет использоваться для чтения данных из файла "output.csv".

CsvReader csvRead = new CsvReader(reader, new CsvConfiguration(CultureInfo.InvariantCulture));
Создание экземпляра CsvReader, который будет использоваться для чтения данных в формате CSV. CsvReader связывается с StreamReader, определенным в предыдущей строке, чтобы читать данные из файла "output.csv".

List<Person> records = csvRead.GetRecords<Person>().ToList();
Чтение данных из файла "output.csv" и преобразование их в список объектов Person. GetRecords<Person>() извлекает строки из CSV файла и автоматически “маппит” их на объекты типа Person. Метод ToList() преобразует результат в список.

foreach (Person person in records) { ... }: Перебор списка объектов Person и отображение каждого объекта на консоль. Этот код отображает имя и возраст каждого человека из списка на консоли.

Тонкая настройка работы.
Когда вы работаете с CSV файлами с использованием библиотеки CSV Helper в C#, вы можете настраивать различные аспекты парсинга и записи данных, такие как разделители, обработка заголовков, атрибуты для классов и маппинг полей.

Настройка разделителя.
По умолчанию CSV Helper использует запятую (,) в качестве разделителя. Однако, вы можете изменить разделитель, если ваши данные используют другой символ для разделения, такой как точка с запятой (;) или табуляция (\t).
Для настройки разделителя, вы можете использовать свойство Delimiter в CsvConfiguration.

Например:

```
// Создание StreamReader для чтения из файла "output.csv".
StreamReader reader = new StreamReader("output.csv");

CsvConfiguration csvConfig = new
CsvConfiguration(CultureInfo.InvariantCulture)
{
    Delimiter = ";"
};

// Создание CsvReader с настройками
CsvReader csvRead = new CsvReader(reader, csvConfig);
```

csvConfig.Delimiter = ";" - Установка разделителя на точку с запятой.

Обработка заголовков.

CSV файлы могут содержать заголовки для каждого столбца, описывающие содержимое столбца.

CSV Helper автоматически использует первую строку файла как заголовки по умолчанию. Если ваши данные не содержат заголовков или вы хотите управлять их обработкой, вы можете использовать опции HasHeaderRecord и HeaderValidated в CsvConfiguration.

Например:

```
CsvConfiguration csvConfig = new
CsvConfiguration(CultureInfo.InvariantCulture)
{
    Delimiter = ";",
    HasHeaderRecord = false, // Если данные не содержат заголовков.
    HeaderValidated = null // Позволяет управлять обработкой
заголовков.
};
```

Атрибуты для классов.

Вы можете использовать атрибуты из пространства имен CsvHelper.Configuration.Attributes для настройки формата данных ваших классов. Например, вы можете использовать атрибут Name для задания пользовательских имен полей, которые будут использоваться при записи в CSV файл.

Мы уже проделывали это при работе с DataGridView, так что тут меняется только атрибут

Пример:

```
public class Person
{
    [Name("UserName")]
    public string Name { get; set; }
    [Name("UserAge")]
    public int Age { get; set; }
}
```

Маппинг полей.

Иногда имена полей в классе могут отличаться от заголовков столбцов в CSV файле. CSV Helper позволяет явно сопоставить имена полей с заголовками столбцов с помощью метода Map в ClassMap.

Например:

```
public sealed class PersonMap : ClassMap<Person>
{
    public PersonMap()
    {
        Map(m => m.Name).Name("Full Name");
        Map(m => m.Age).Name("Age");
    }
}
```

Давайте подробно разберем, как это работает:

- public sealed class PersonMap : ClassMap<Person>: Мы создаем класс PersonMap, который наследуется от ClassMap<T>, где T - это тип данных, который вы хотите сопоставить с CSV файлом. В данном случае, это класс Person.
- В конструкторе PersonMap(), мы определяем маппинг полей класса Person на заголовки столбцов в CSV файле.
- Map(m => m.Name).Name("Full Name"): Это указывает, что поле Name класса Person соответствует столбцу с заголовком "Full Name" в CSV файле. То есть, при чтении данных из файла, CSV Helper будет искать столбец с именем "Full Name" и присваивать его значение полю Name в объекте Person.

- Map(m => m.Age).Name("Age"): Аналогично, это указывает, что поле Age класса Person соответствует столбцу с заголовком "Age" в CSV файле.

Теперь, при использовании этого маппинга, при чтении данных из CSV файла с использованием CsvReader, CSV Helper будет использовать указанные маппинги, чтобы правильно сопоставить данные из столбцов с соответствующими полями объекта Person.
При записи данных в CSV файл с использованием CsvWriter, маппинги также будут использоваться для форматирования данных и создания столбцов с указанными заголовками.

Этот маппинг позволяет гибко настраивать соответствие между полями вашего класса и столбцами CSV файла, что особенно полезно, если имена полей и заголовков в CSV файле отличаются.

Используя эти конструкции, вы можете настроить и оптимизировать работу CSV Helper для вашей конкретной задачи и формата данных.

CSV Helper также предоставляет множество других возможностей. Вы можете изучить официальную документацию для более подробной информации и примеров использования:
https://joshclose.github.io/CsvHelper/