Практическая работа № 2.  
Разработка Windows Forms приложений  
для сериализации данных

**Цель практической работы**

Закрепление теоретических знаний по созданию Windows Forms приложения для сериализации данных в JSON.

**Постановка задачи**

**Сериализация** — это процесс **преобразования объекта** в поток байтов (в нашем случае **в строку**) 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**.

Например, мы хотим создать объект типа Employee и сохранить его в файл, чтобы потом можно было загрузить с заполненными данными.

public class Employee

{

public string FirstName { get; set; } = "";

public string LastName { get; set; } = "";

public double Salary { get; set; }

public Employee(

string firstName,

string lastName,

double salary)

{

FirstName = firstName;

LastName = lastName;

Salary = salary;

}

}

Для начала нужно скачать с помощью NuGet пакетов библиотеку **Newtonsoft.JSON** (рисунок 1).

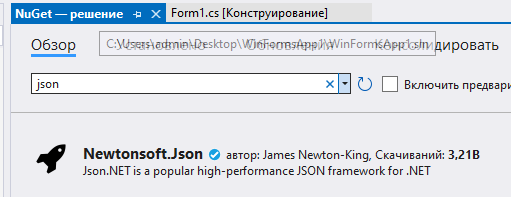


Рисунок 1 – Установка библиотеки Newtonsoft.Json

Создаем объект сотрудника, и сериализуем в строку формата JSON, далее сохраняем в файл стандартным способом:

var employee = new Employee("John", "Black", 3000);

var jsonString = JsonConvert.SerializeObject(employee);

File.WriteAllText("file.json", jsonString);

В данном случае **сериализуются все публичные поля и свойства!**

Полученный файл представлен на рисунке 2. Данный файл можно открыть любым текстовым редактором, также удобно использовать **VisualStudio** или **Notepad++**.

Данные по сотруднику также можно было сохранить в виде текста вручную: сформировать самим строку определенного формата и сохранить ее, однако тогда нужно было бы продумать обратную операцию получения объекта из строки. Очевидно, что гораздо удобнее пользоваться готовым решением в виде JSON.

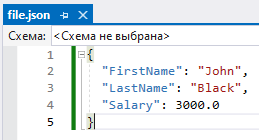


Рисунок 2 – Содержимое JSON файла

**Для десериализации** (получения объекта из JSON строки) нужно написать:

var employee1 = JsonConvert.DeserializeObject<Employee>(jsonString);

Далее рассмотрим, **что делать, если нужно сериализовать приватное поле или свойство, или наоборот если что-то сериализовать не нужно.**

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные **метаданные**.

**Атрибуты указываются в квадратных скобках**, над классом, полем или свойством.

Чтобы явно указать, **что объект можно сериализовать** используется атрибут [Serializable] (для рассматриваемой библиотеки он не обязателен).

Если мы хотим задавать свои имена у данных **в файле JSON** нужно использовать.

[DataContract] перед именем класса, а сами поля и свойства пометить [DataMember].

Чтобы отметить нужный конструктор для десериализации, нужно использовать атрибут [JsonConstructor], при этом имена переменных в конструкторе должны совпадать с именами соответствующих полей и свойств (без учета регистра).

[DataContract]

public class Employee

{

[DataMember(Name = "Имя")]

public string FirstName { get; set; } = "";

[DataMember(Name = "Фамилия")]

public string LastName { get; set; } = "";

[DataMember(Name = "Зарплата")]

public double Salary { get; set; }

[JsonConstructor]

public Employee(string firstName, string lastName, double salary)

{

FirstName = firstName;

LastName = lastName;

Salary = salary;

}

}

На рисунке 3 представлен результат сериализации объекта с заданными именами полей.

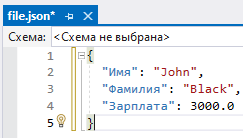


Рисунок 3 – Содержимое JSON файла  
с заданными именами полей

Чтобы указать, что поле не нужно сериализовать необходимо отметить его атрибутом [JsonIgnore], результат представлен на рисунке 4.

[DataContract]

public class Employee

{

[JsonIgnore]

public string FirstName { get; set; } = "";

[DataMember(Name = "Фамилия")]

public string LastName { get; set; } = "";

[DataMember(Name = "Зарплата")]

public double Salary { get; set; }

[JsonConstructor]

public Employee(string firstName, string lastName, double salary)

{

FirstName = firstName;

LastName = lastName;

Salary = salary;

}

}

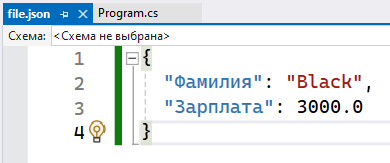


Рисунок 4 – Содержимое JSON файла  
с исключением поля «FirstName»

Также можно сериализовать вложенные классы. Пример сериализации объекта класса, содержащего внутри себя другие классы:

[DataContract]

public class Company

{

[DataMember(Name = "Директор")]

public Employee Director { get; set; }

[DataMember(Name = "Сотрудники")]

public List<Employee> Employees { get; set; } = new();

[JsonConstructor]

public Company(Employee director, List<Employee> employees)

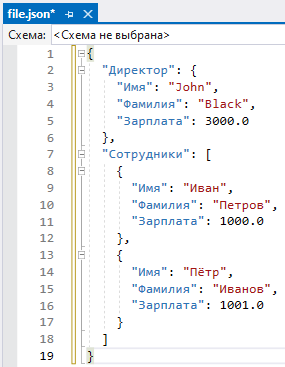
{

Director = director;

Employees = employees;

}

}



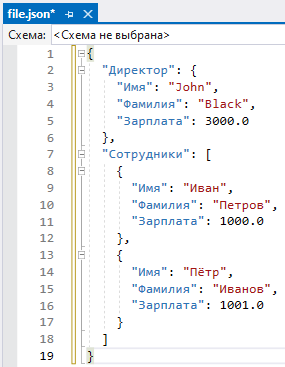


Рисунок 5 – Содержимое JSON файла с вложенными классами

Если объекты имеют сложную и запутанную структуру, являются наследниками класса, требуют вызова каких-либо метода для инициализации, то автоматическая сериализация будет затруднена или невозможна. Рассмотрим пример «ручной» сериализации.

var employee = new Employee("John", "Black", 3000);

JObject j = new JObject();

j["Имя"] = employee.FirstName;

j["Фамилия"] = employee.LastName;

j["Зарплата"] = employee.Salary;

var status = employee.Salary switch

{

< 100 => "Бедняк",

< 1000 => "Средний класс",

< 5000 => "Богач",

\_ => "Сказочный богач"

};

j["Статус"] = status;

File.WriteAllText("file.json", j.ToString());

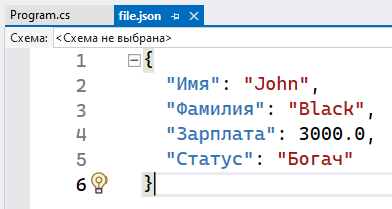


Рисунок 6 – Содержимое JSON файла после ручного заполнения

Видно, что работа с JSON напоминает работу со словарем. напишем код для чтения данного файла.

var j\_str = File.ReadAllText("file.json");

var json = JObject.Parse(j\_str);

var employee1 = new Employee(

json["Имя"].Value<string>(),

json["Фамилия"].Value<string>(),

json["Зарплата"].Value<int>());

**Регулярные выражения** представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками. Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex.

Например, у нас есть текст "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и нам надо найти в нем все словоформы «туп»:

using System.Text.RegularExpressions;

var str = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

var regex = new Regex(@"туп\w\*"); //Задаем шаблон:

var matches = regex.Matches(str);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else Console.WriteLine("Совпадений не найдено");

В данном случае последовательность начинается с «туп», \w означает любой символ слова (любые буквы, а также знаки и цифры, если они являются частью слова, например, «А.13» будет считаться словом и все её символы будут подходить по шаблону \w).

Знак звездочки «\*» означает, что выражение левее него может повторяться 0 раз и более. В данном случае это относится к \w.

Класс символов соответствует любому из набора символов. Классы символов включают языковые элементы, перечисленные в следующей таблице.

|  |  |  |  |
| --- | --- | --- | --- |
| **Класс** | **Описание** | **Шаблон** | **Ответ** |
| [ группа\_символов ] | Единичный символ из группы (чувствителен к регистру) | [ae] | "a" в "gray" "a", "e" в "lane" |
| [^ группа\_символов ] | НЕ единичный символ из группы (инверсия предыдущего примера) | [^aei] | "r", "g", "n" в "reign" |
| [ первый - последний ] | Диапазон символов | [A-Z] | "A", "B" в "AB123" |
| . (Знак точки) | Любой единичный символ | a.e | "ave" в "nave" "ate" в "water" |
| \p{ кодировка } | Любой единичный символ заданной кодировки | \p{Lu}  \p{IsCyrillic} | "C", "L" в "City Lights"  "Д", "Ж" в "ДЖem" |
| \s | Любой символ пробела | \w\s | "D " в "ID A1.3" |

Квантификаторы определяют количество экземпляров символа, группы или класса символов, которое должно присутствовать во входных данных, чтобы было зафиксировано совпадение. В следующей таблице перечислены квантификаторы, поддерживаемые .NET:

|  |  |  |  |
| --- | --- | --- | --- |
| **Квантор** | **Описание** | **Шаблон** | **Ответ** |
| \* | Предыдущий элемент повторяется 0 и более раз | a.\*c | "abcbc" в "abcbc" |
| + | 1 и более раз | "be+" | "bee" в "been", "be" в "bent" |
| ? | 0 или 1 раз | "rai?" | "rai" в "rain" |
| { *n* } | n раз | ",\d{3}" | ",043" в "1,043.6", ",876", ",543", и ",210" в "9,876,543,210" |
| { *n* ,} | n и более раз | "\d{2,}" | "166", "29", "1930" |
| { *n* , *m* } | от n до m раз включительно | "\d{3,5}" | "166", "17668" "19302" в "193024" |

**В жадном режиме** (по умолчанию) квантификатор повторяется столько раз, сколько это возможно.

Движок регулярного выражения пытается получить максимальное количество символов, соответствующих .+, а затем сокращает это количество символ за символом, если остаток шаблона не совпадает. Далее рассмотрим не жадные кванторы.

|  |  |  |  |
| --- | --- | --- | --- |
| **Квантор** | **Описание** | **Шаблон** | **Ответ** |
| \*? | Предыдущий элемент повторяется 0 и более раз | a.\*?c | "abc" в "abcbc" |
| +? | 1 и более раз | "be+?" | "be" в "been", "be" в "bent" |
| ?? | 0 или 1 раз | "rai??" | "ra" в "rain" |
| { *n* }? | n раз | ",\d{3}?" | ",043" в "1,043.6", ",876", ",543", и ",210" в "9,876,543,210" |
| { *n* ,}? | n и более раз | "\d{2,}?" | "166", "29", "1930" |
| { *n* , *m* }? | от n до m раз включительно | "\d{3,5}?" | "166", "17668"  "193", "024" в "193024" |

Привязки (или атомарные утверждения нулевой ширины) указывают положение в строке, где должно быть найдено соответствие. При использовании привязки в выражении поиска обработчик регулярных выражений не проходит по строке и не потребляет символы; он ищет соответствия только в заданном местоположении. Например, ^ указывает, что соответствие должно начаться в начале строки. Таким образом, регулярное выражение ^http: находит соответствие для http, только если этот элемент находится в начале строки. В таблице ниже перечислены привязки, поддерживаемые регулярными выражениями в .NET.

|  |  |  |  |
| --- | --- | --- | --- |
| **Привязка** | **Описание** | **Шаблон** | **Ответ** |
| ^ | Начало строки или строк в случае многострочного вывода. | ^\d{3} | "901" в "901-333-" |
| $ | Конец строки или строк в случае многострочного вывода. | -\d{3}$ | "-333" в "-901-333" |
| \A | Начало строки. | \A\d{3} | "901" в "901-333-" |
| \Z | Конец строки или до последнего символа новой строки. | -\d{3}\Z | "-333" в "-901-333" |
| \z | Конец строки. | -\d{3}\z | "-333" в "-901-333" |
| \G | Непрерывное соответствие. | \G\(\d\) | "(1)", "(3)", "(5)"  в "(1)(3)(5)[7](9) |
| \b | На границе слова | \b\w+\s\w+\b | "them theme", "them them" в "them theme them them" |

**Параметр RegexOptions.**

Regex regex = new Regex(@"туп(\w\*)",RegexOptions.IgnoreCase);

Regex regex = new Regex(@"туп(\w\*)",

RegexOptions.Compiled | RegexOptions.IgnoreCase);

**Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение

**CultureInvariant**: при установке этого значения будут игнорироваться региональные различия

**IgnoreCase**: при установке этого значения будет игнорироваться регистр

**IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #

**Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста

**RightToLeft**: приписывает читать строку справа налево

**Singleline**: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

**Пример** поиска номера в формате 111-111-1111:

string s = "456-435-2318";

Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}");

Перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

string s = "456-435-2318";

Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");

С помощью операции | можно задать альтернативные символы, например:

Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}");

Первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

**Проверка** на соответствие строки формату, метод **IsMatch**.

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод IsMatch, который позволяет проверить входную строку с шаблоном на соответствие:

var strings = new[] { "example@mail.ru", "examplemail.ru", "example@mailru" };

Regex regex = new Regex(@"\w+@\w+\.\w+");

foreach (string s in strings)

if (regex.IsMatch(s))

Console.WriteLine(s);

**Замена символов, метод Replace.**

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

string text = "Мама мыла раму. ";

string pattern = @"\s+";

string target = " ";

Regex regex = new Regex(pattern);

string result = regex.Replace(text, target);

Console.WriteLine(result);

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов одинарными.

Рефлексия — это мощная возможность в .NET, позволяющая программам исследовать свои собственные структуры данных и метаданные во время выполнения. С помощью рефлексии можно получать информацию о типах, методах, свойствах и других членах классов, а также динамически создавать экземпляры объектов и вызывать методы.

**Основные понятия рефлексии**

1. **Типы**: В C# типы могут быть классами, структурами, интерфейсами и перечислениями. Рефлексия позволяет исследовать все эти типы.
2. **Метаданные**: Это информация о типах, которая хранится в сборках (assemblies) .NET. Метаданные включают информацию о свойствах, методах, атрибутах и других членах типа.
3. **Assembly**: Это сборка, которая содержит код, метаданные и ресурсы. С помощью рефлексии можно загружать сборки и получать информацию о них.

Рефлексия может использоваться для различных целей, таких как:

* Динамическое создание объектов
* Вызываем методов по имени
* Получение и установка значений свойств
* Создание инструментов для сериализации и десериализации данных
* Реализация механизмов фильтрации и поиска

**Пример использования рефлексии: Сериализация и десериализация объектов**

В этом примере мы создадим приложение для сериализации и десериализации объектов в формате JSON. Мы будем использовать рефлексию для динамического получения значений свойств объектов и их установки.

**Шаг 1: Определение класса Person**

Создадим класс Person, который будет представлять человека с несколькими свойствами.

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public string Email { get; set; }

public override string ToString()

{

return $"Name: {Name}, Age: {Age}, Email: {Email}";

}

}

**Шаг 2: Реализация метода сериализации**

Создадим метод Serialize, который будет принимать объект и возвращать его представление в формате JSON.

using System.Text;

public static class JsonSerializer

{

public static string Serialize(object obj)

{

var type = obj.GetType();

var properties = type.GetProperties();

var jsonBuilder = new StringBuilder();

jsonBuilder.AppendLine("{");

var i = 0;

foreach (var property in properties)

{

var value = property.GetValue(obj);

jsonBuilder.Append($"\"{property.Name}\": \"{value}\"");

if (i < properties.Length - 1)

jsonBuilder.AppendLine(",");

i++;

}

// Удаляем последнюю запятую

if (jsonBuilder.Length > 1)

jsonBuilder.Length--;

jsonBuilder.AppendLine();

jsonBuilder.AppendLine("}");

return jsonBuilder.ToString();

}

}

**Шаг 3: Реализация метода десериализации**

Создадим метод Deserialize, который будет принимать строку JSON и возвращать объект заданного типа.

public static class JsonDeserializer

{

public static object Deserialize(Type type, string json)

{

object? obj = Activator.CreateInstance(type);

var properties = type.GetProperties();

// Удаляем фигурные скобки и разбиваем по запятой

var pairs = json.Trim('{', '}').Split(',');

foreach (var pair in pairs)

{

var keyValue = pair.Split(':');

var propertyName = keyValue[0].Trim().Trim('"');

var propertyValue = keyValue[1].Trim().Trim('"');

var property = type.GetProperty(propertyName);

if (property != null)

{

// Преобразуем строковое значение в тип свойства

if (property.PropertyType == typeof(int))

{

property.SetValue(obj, int.Parse(propertyValue));

}

else

{

property.SetValue(obj, propertyValue);

}

}

}

return obj;

}

}

**Шаг 4: Тестирование сериализации и десериализации**

Теперь протестируем наши методы в Main.

class Program

{

static void Main(string[] args)

{

// Создаем объект Person

var person = new Person

{

Name = "Alice",

Age = 30,

Email = "alice@example.com"

};

// Сериализуем объект в JSON

string json = JsonSerializer.Serialize(person);

Console.WriteLine("Сериализованный JSON:");

Console.WriteLine(json);

// Десериализуем JSON обратно в объект

var deserializedPerson = JsonDeserializer.Deserialize(typeof(person), json);

Console.WriteLine("\nДесериализованный объект:");

Console.WriteLine(deserializedPerson);

}

}

**Запуск приложения**

Когда вы запустите приложение, вы увидите вывод, который показывает сериализованный JSON и десериализованный объект:

Сериализованный JSON:

{"Name": "Alice","Age": "30","Email": "alice@example.com"}

Десериализованный объект:

Name: Alice, Age: 30, Email: alice@example.com

**Пример создания плагина с использованием позднего связывания.**

1. Сначала создадим интерфейс для плагинов (проект №1):

// IPlugin.cs

public interface IPlugin

{

    void Execute();

}

1. Затем создадим класс плагина, который реализует этот интерфейс (проект №2, подключаем проект №1 с помощью добавления зависимостей):

// MyPlugin.cs

public class MyPlugin : IPlugin

{

    public void Execute()

    {

        Console.WriteLine("MyPlugin executed!");

    }

}

1. Теперь создадим основной класс, который будет загружать сборку плагина и вызывать метод Execute (проект №3, подключаем проект №1 с помощью добавления зависимостей):

// Program.cs

using System;

using System.IO;

using System.Reflection;

class Program

{

    static void Main(string[] args)

    {

        // Путь к сборке плагина

        string pluginPath = "MyPlugin.dll"; // Убедитесь, что этот файл существует в директории

        // Загружаем сборку плагина

        Assembly pluginAssembly = Assembly.LoadFrom(pluginPath);

        // Ищем типы, реализующие интерфейс IPlugin

        foreach (Type type in pluginAssembly.GetTypes())

        {

            if (typeof(IPlugin).IsAssignableFrom(type) && !type.IsInterface && !type.IsAbstract)

            {

                // Создаем экземпляр плагина

                IPlugin pluginInstance = (IPlugin)Activator.CreateInstance(type);

                // Вызываем метод Execute

                pluginInstance.Execute();

            }

        }

    }

}

**Варианты заданий на практическую работу**

1. Создать Windows Forms приложение для регистрации.
2. Добавить проверку введенных данных на соответствие шаблонам с помощью регулярных выражений.
3. Регистрационная форма должна включать:

* Логин,
* Пароль,
* Электронная почта.

1. Данные пользователя необходимо сериализовать и записать в файл формата JSON.
2. Сериализацию выполнить с помощью рефлексии.

Указания к работе

**1. Обязательные поля**

**1.1 Логин**

**Требования:**

* Разрешены **только латинские буквы (a-z, A-Z) и цифры (0-9)**.
* **Длина:** от 5 до 20 символов.
* **Не может начинаться с цифры**.
* **Запрещены спецсимволы** (кроме подчеркивания \_).

**Примеры:**

✅ User123, Admin\_Test

❌ 123User, !Login, User@

**1.2 Пароль**

**Требования:**

* **Минимум 8 символов.**
* **Обязательно:**
  + хотя бы **1 заглавная буква (A-Z)**,
  + хотя бы **1 цифра (0-9)**,
  + хотя бы **1 спецсимвол** (!@#$%^&\* и т. д.).
* **Запрещены пробелы.**
* **Не может состоять только из цифр.**

**Примеры:**  
✅ Pass123!, Secure#2024

❌ password, 12345678, Pass 123

**1.3 Электронная почта**

**Требования:**

* **Стандартный формат:** username@domain.tld.
* **Допустимые символы:**
  + Латиница (a-z, A-Z), цифры (0-9), точки ., подчеркивание \_, дефис -.
* **Домен:**
  + Минимум **2 символа** после точки (.com, .ru).
  + Запрещены **disposable-email** (типа temp-mail.org).

**Примеры:**

✅ user@example.com, [john.doe@mail.ru](mailto:john.doe@mail.ru)

❌ user@.com, user@temp-mail.org, user@site

**Пример разработки приложения**

**Корабль**

* Name – Название
* Type – Тип (пассажирский/грузовой)
* DisplacementTons – Водоизмещение (тонн)
* MaxSpeedKnots – Макс. скорость (узлов)
* IsMilitary – Военный/гражданский

**Шаг 1: Создание проекта Windows Forms**

1. Откройте Visual Studio.
2. Создайте новый проект: **Windows Forms App (.NET Framework)**.
3. Назовите проект, например, ShipApp.

**Шаг 2: Добавление класса Ship**

1. Правой кнопкой по проекту → **Add** → **Class**.
2. Назовите класс Ship.cs.
3. Реализуйте класс:

using System.Runtime.Serialization;

namespace WinFormsApp1;

public class Ship

{

public Ship(

string name,

string type,

double displacementTons,

double maxSpeedKnots,

bool isMilitary)

{

Name = name;

Type = type;

DisplacementTons = displacementTons;

MaxSpeedKnots = maxSpeedKnots;

IsMilitary = isMilitary;

}

[DataMember(Name = "Название")]

public string Name { get; set; }

[DataMember(Name = "Тип")]

public string Type { get; set; } // "Пассажирский" или "Грузовой"

[DataMember(Name = "Водоизмещение, тонн")]

public double DisplacementTons { get; set; }

[DataMember(Name = "Макс. скорость, узлов")]

public double MaxSpeedKnots { get; set; }

[DataMember(Name = "Военный?")]

public bool IsMilitary { get; set; }

}

**Шаг 3: Добавление Newtonsoft.Json для работы с JSON**

1. Правой кнопкой по проекту → **Manage NuGet Packages**.
2. Найдите и установите Newtonsoft.Json.

**Шаг 4: Дизайн формы (Form1)**

1. Откройте Form1 в дизайнере.
2. Добавьте элементы управления:
   * **TextBox** для Name (название).
   * **ComboBox** для Type (тип корабля).
   * **NumericUpDown** для DisplacementTons и MaxSpeedKnots.
   * **CheckBox** для IsMilitary (военный/гражданский).
   * **Button** для сохранения (btnSave).
   * **Button** для загрузки (btnLoad).
   * **Label** для вывода информации (lblInfo).

Примерный дизайн представлен на рисунке 7.

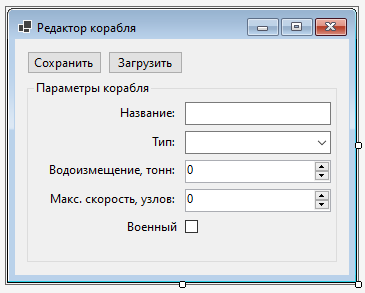


Рисунок 7 – Дизайн формы

**Шаг 5: Код формы (Form1.cs)**

using Newtonsoft.Json;

namespace WinFormsApp1;

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

// Заполняем ComboBox типами кораблей

comboBoxType.Items.AddRange(new[] { "Пассажирский", "Грузовой" });

comboBoxType.SelectedIndex = 0;

}

private void btnSave\_Click(object sender, EventArgs e)

{

var ship = new Ship(

textBoxName.Text,

comboBoxType.SelectedItem.ToString(),

(double)numericUpDownDisplacement.Value,

(double)numericUpDownMaxSpeed.Value,

checkBoxIsMilitary.Checked);

string json = JsonConvert.SerializeObject(ship, Formatting.Indented);

var sfd = new SaveFileDialog

{

DefaultExt = ".json",

Filter = "JSON | \*.json"

};

if (sfd.ShowDialog() != DialogResult.OK)

return;

File.WriteAllText(sfd.FileName, json);

MessageBox.Show($"Данные сохранены в {sfd.FileName}");

}

private void btnLoad\_Click(object sender, EventArgs e)

{

var ofd = new OpenFileDialog

{

DefaultExt = ".json",

Filter = "JSON | \*.json"

};

if (ofd.ShowDialog() != DialogResult.OK)

return;

if (!File.Exists(ofd.FileName))

MessageBox.Show($"Файл {ofd.FileName} не найден!");

string json = File.ReadAllText(ofd.FileName);

Ship ship = JsonConvert.DeserializeObject<Ship>(json);

textBoxName.Text = ship.Name;

comboBoxType.SelectedItem = ship.Type;

numericUpDownDisplacement.Value = (decimal)ship.DisplacementTons;

numericUpDownMaxSpeed.Value = (decimal)ship.MaxSpeedKnots;

checkBoxIsMilitary.Checked = ship.IsMilitary;

}

}

**Шаг 6: Инициализация компонентов (Form1.Designer.cs)**

Убедитесь, что в Form1.Designer.cs корректно объявлены все элементы управления (они создаются автоматически при перетаскивании из Toolbox).

**Шаг 7: Тестирование приложения**

1. Запустите приложение (**F5**).
2. Введите данные:
   * Название: "Титаник".
   * Тип: "Пассажирский".
   * Водоизмещение: 52310.
   * Макс. скорость: 23.
   * Военный: Нет.
3. Нажмите **Сохранить** → выберите куда сохранить файл.
4. Нажмите **Загрузить** → выберите откуда загрузить файл, данные появятся в форме.

# Содержание пояснительной записки

1. Постановка задачи. Приводится теоретический материал, использованный при написании приложения.

2. Формулировка задания и вариант. Приводится задание на лабораторную работу и вариант этого задания.

3. Описание выполняемых действий. Необходимо привести описание последовательности разработки программы, реализации используемых методов, алгоритмов, блок-схем.

4. Анализ результатов. Привести анализ входных и выходных данных. Показать результаты выполнения программного кода. Предоставить скриншоты обработки тестовых примеров. Сделать выводы.

5. Листинг программы. Привести листинг разработанного программного кода, содержание файлов входных и выходных данных.

# Используемое программное обеспечение

1. Среда программирования MS Visual Studio Community 2022 (Свободно распространяемое программное обеспечение (в учебных целях));
2. Microsoft Office Standard 2007 (Open License: 42267924);
3. Open Office (Свободно распространяемое программное обеспечение).
4. Браузер (Свободно распространяемое программное обеспечение).

# Список литературы

* + - 1. Мейер Б. Объектно-ориентированное программирование и программная инженерия [Электронный ресурс]/ Мейер Б. – Электрон. текстовые данные. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 285 c.
      2. Биллиг, В. A. Основы объектного программирования на С# (C# 3.0, Visual Studio 2008) [Электронный ресурс] : учебное пособие / В. A. Биллиг. — Электрон. текстовые данные. — Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017. — 583 c. — 978-5-4487-0145-0. — Режим доступа: http://www.iprbookshop.ru/72339.html
      3. Павловская, Т. А. Программирование на языке высокого уровня C# [Электронный ресурс] / Т. А. Павловская. — 2-е изд. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 245 c. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/73713.html
      4. Агапов, В. П. Основы программирования на языке С# [Электронный ресурс] : учебное пособие / В. П. Агапов. — Электрон. текстовые данные. — М. : Московский государственный строительный университет, ЭБС АСВ, 2012. — 128 c. — 978-5-7264-0576-6. — Режим доступа: http://www.iprbookshop.ru/16366.html
      5. Медведев, М. А. Программирование на СИ# [Электронный ресурс] : учебное пособие / М. А. Медведев, А. Н. Медведев ; под ред. А. В. Присяжный. — Электрон. текстовые данные. — Екатеринбург : Уральский федеральный университет, ЭБС АСВ, 2015. — 64 c. — 978-5-7996-1561-1. — Режим доступа: http://www.iprbookshop.ru/69667.html
      6. Казанский А.А. Объектно-ориентированное программирование на языке Microsoft Visual С# в среде разработки Microsoft Visual Studio 2008 и .NET Framework. 4.3 [Электронный ресурс]: учебное пособие и практикум/ Казанский А.А.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2011.— 180 c
      7. Уйманова Н.А. Основы объектно-ориентированного программирования [Электронный ресурс]: практикум/ Уйманова Н.А., Таспаева М.Г.— Электрон. текстовые данные.— Оренбург: Оренбургский государственный университет, ЭБС АСВ, 2017.— 156 c.— Режим доступа: http://www.iprbookshop.ru/78808.html.— ЭБС «IPRbooks»
      8. Новиков П.В. Объектно-ориентированное программирование [Электронный ресурс]: учебно-методическое пособие к лабораторным работам/ Новиков П.В.— Электрон. текстовые данные.— Саратов: Вузовское образование, 2017.— 124 c.— Режим доступа: http://www.iprbookshop.ru/64650.html.— ЭБС «IPRbooks»