# Лабораторная работа №4: Интерфейсы.

**Цель работы:**

– Знакомство с основными принципами ООП.

**Теоретическая информация.**

**Интерфейс** - это абстрактный класс, который определяет поведение и свойства для других классов. В C# интерфейсы используются для определения набора методов, свойств, событий, которые должны быть реализованы в классе, импортирующем этот интерфейс.

Пример создания интерфейса:

```
//интерфейс для логгирования
public interface ILogger
    {
        void Log(string message);
        void Error(string errorMessage);
        string Name { get; }
    }
```

Так же, как и наследования для реализации интерфейса используется указатель «:» перед классом, после чего идет имя интерфейса:

```
//Класс, реализующий методы интерфейса
public class ConsoleLogger : ILogger
    {
        public void Log(string message)
        {
            Console.WriteLine(message);
        }

        public void Error(string errorMessage)
        {
            Console.WriteLine("ERROR: " + errorMessage);
        }

        public string Name => "Console Logger";
    }
```

В этом примере класс ConsoleLogger реализует интерфейс ILogger, что означает, что он должен иметь методы Log и Error, а также свойство Name.

Так же, как и абстрактные классы можно считать, что каждый метод и свойство интерфейса являются «абстрактными» - это значит, что каждый метод и свойство должны быть переопределены и реализовывать некий функционал в классе реализующим интерфейс.

Одной из важных особенностей интерфейсов является возможность реализации нескольких интерфейсов классом:

```
//интерфейсы работы с консолью
public interface ILogger
    {
        void Log(string message);
        void Error(string errorMessage);

        string Name { get; }
    }

public interface IWriter
    {
        void Write(string text);
        void WriteLine(string text);
    }

public interface IPrinter
    {
        void Print(string text);
    }
```

Интерфейсы, реализуемые классом, перечисляются через запятую:

```
//Класс реализующий все интерфейсы
public class Printer : ILogger, IWriter, IPrinter
    {
        public void Log(string message)
        {
            Console.WriteLine("LOG: " + message);
        }

        public void Error(string errorMessage)
        {
            Console.WriteLine("ERROR: " + errorMessage);
        }

        public string Name => "Printer";

        public void Write(string text)
        {
            Console.Write(text);
        }

        public void WriteLine(string text)
        {
            Console.WriteLine(text);
        }

        public void Print(string text)
        {
            Console.WriteLine(text);
        }
    }
```

Реализация интерфейсов не запрещает использовать наследование – класс все также может одновременно наследоваться от одного класса и реализовывать один и более интерфейсов, главное, чтобы класс от которого происходит наследования стоял первым после знака наследования.

Это пример использования принципа "одного наследования и множества реализации" (Single Responsibility Principle), который гласит, что класс должен иметь только одну основную задачу которая описана в классе или при наследовании, но также может иметь несколько второстепенных обязанностей реализуемых интерфейсами.

Преимущества такого подхода:

- Классы имеют четкую ответственность и не перегружаются множеством обязанностей.
- Логика разделена между классами, что делает ее более гибкой и модульной.
- Новую логику можно легко добавить без изменения существующего кода.

**Задание №1. Реорганизация редактора противников для использования интерфейсов. Создание новых типов противников.**

Реорганизовать основной функционал программы в соответствии с задачами.

**Функционал программы:**

- К основному функционалу программы редактора добавлен функционал создания нескольких типов противников.

**Задачи:**

- Реализовать добавление нескольких типов противников.
- Сохранение и загрузка листа противников происходит с использованием интерфейсов.

Создадим новый тип противника, имеющий параметр брони. Перед этим поменяем класс CEnemyTemplate на абстрактный и поменяем связку атрибут-функция на атрибут-свойство в классе:

```
public abstract class CEnemyTemplate
    {
        //Вместо функции для получения данных используется свойсвто
        string name;
        public string Name
        {
            get { return name; }
            set { if (value.Length > 0) name = value; else name = "unknown"; }
        }

    }
```

Таким же способом требуется переопределить все остальные атрибуты класса. Использование свойств облегчит процесс сохранения и загрузки параметров в формат JSON.

Затем на основе этого абстрактного класса создадим класс обычного противника, а так же класс бронированного противника имеющего свойство брони:

```
//Обычный противник
    public class CNormalEnemyTemplate : CEnemyTemplate { }

//Противник со свойством брони
public class CArmoredEnemyTemplate : CEnemyTemplate
    {
        double armor;
        public double Armor
        {
            get { return armor; }
            set { if (value > 0) armor = value; else armor = 25; }
        }
    }
```

Таким образом требуется создать еще два вида противников: укорачивающийся противник и противник, который лечится с определенным шансом при получении урона. Так же требуется реализовать изменение интерфейса программы и логики добавления противников в список.

Так как библиотеки, работающие с JSON на версиях до .NET 7 не имеют встроенную поддержку полиморфизма потребуется написать свой сериализатор. Для этого требуется создать новый класс, который будет наследоваться от класса **JsonConverter :**

```
//Класс конвертера используется в связке с родительским классом шаблона противника
public class EnemyTemplateConverter : JsonConverter<CEnemyTemplate>
{

}
```

Сперва требуется переопределить функцию чтения Read, ответственную за чтение и десериализацию json-документа. В данной функции будет определятся к какому классу непосредственно относится считываемый документ:

```
public override CEnemyTemplate Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
    {
        // Создаем JSON-документ
        var jsonDoc = JsonDocument.ParseValue(ref reader);

        try
        {
            // Получаем тип объекта
            string type = jsonDoc.RootElement.GetProperty("$type").GetString();

            switch (type)
            {
                //Определение типа бронированного противника
                case "CArmoredEnemyTemplate":
                    return
JsonSerializer.Deserialize<CArmoredEnemyTemplate>(jsonDoc.RootElement.GetRawText(), options);
                //Определение типа обычного противника
                case "CNormalEnemyTemplate":
                    return
JsonSerializer.Deserialize<CNormalEnemyTemplate>(jsonDoc.RootElement.GetRawText(), options);
                default:
                    throw new NotSupportedException($"Unknown type: {type}");
            }
        }
        finally
        {
            // Освобождаем ресурс
            jsonDoc.Dispose();
        }
    }
```

Реализуя новых противников не забудьте добавить их в блок switch для того чтобы сериализатор имел возможность корректно определить класс.

Второй функцией, которую требуется переопределить будет функция записи Write. В данном случае будет добавлен функционал для записи информации о том к какому классу относится сохраняемый документ:

```
public override void Write(Utf8JsonWriter writer, CEnemyTemplate value, JsonSerializerOptions
options)
    {
        string type = value.GetType().Name;  // Определяем тип: CNormalEnemyTemplate,
CArmoredEnemyTemplate и т. д.

        string json = JsonSerializer.Serialize(value, value.GetType(), options);

        var jsonDoc = JsonDocument.Parse(json);

        try
        {
            writer.WriteStartObject();
            writer.WriteString("$type", type);  // Добавляем информацию о типе

            // Копируем все свойства
            foreach (var property in jsonDoc.RootElement.EnumerateObject())
            {
                property.WriteTo(writer);
            }

            writer.WriteEndObject();
        }
        finally
        {
            jsonDoc.Dispose();  // Освобождаем ресурс
        }
    }
```

Создадим интерфейс, имеющий две сигнатуры – сигнатура загрузки и сигнатура сохранения листа от универсального параметра:

```
public interface ISaveList<T>
    {
        //Сигнатура загрузки
        T Load(string path);
        //Сигнатура сохранения
        void Save(T data, string path);
    }
```

Далее создадим класс JsonEnemySaver, который будет реализовывать интерфейс сохранения списка противников в формат Json. Внутри класса содержатся опции в которых в качестве конвертера зададим конвертер который был реализован ранее:

```
public class JsonEnemySaver: ISaveList<List<CEnemyTemplate>>
    {

        private readonly JsonSerializerOptions _options;

        public JsonEnemySaver()
        {
            _options = new JsonSerializerOptions
            {
                WriteIndented = true,
                //Установка конвертера противников для реализации полиморфизма
                Converters = { new EnemyTemplateConverter() }
            };
        }

    }
```

-Так как интерфейс является абстрактным классом требуется обязательно реализовать сигнатуры определённые в интерфейсе:

```
//Реализация функции загрузки
public List<CEnemyTemplate> Load(string path)
        {
            if (File.Exists(path))
            {
                string json = File.ReadAllText(path);

                //Десериализация с определение класса противника
                return JsonSerializer.Deserialize<List<CEnemyTemplate>>(json, _options) ?? new
List<CEnemyTemplate>();
            }

            return new List<CEnemyTemplate>();
        }

//Реализация функции сохранения
public void Save(List<CEnemyTemplate> data, string path)
        {
            string json = JsonSerializer.Serialize(data, _options);
            File.WriteAllText(path, json);
        }
```

После этого в классе CEnemyTemplateList можно добавить реализацию данного сериализатора и функции сохранения и загрузки переопределить для использования функций самого сериализатора:

```
public class CEnemyTemplateList
    {
        private readonly ISaveList<List<CEnemyTemplate>> _serializer = new JsonEnemySaver();

    }
```

**Задание №2. Реорганизация игры-кликкера для использования новых типов противников.**

Разработать основной функционал программы, в которой взаимодействие пользователя с базой данных магазина реализовано при помощи классов.

**Функционал программы:**

- К оригинальным обычным противникам добавлены противники с эффектами.
- Система сохранения прогресса.

**Задачи:**

- Реализовать взаимодействие с противниками разных типов в едином интерфейсе.
- Загрузка листа противников происходит с использованием интерфейса.
- Сохранение и загрузка прогресса происходит с использованием ранее реализованного интерфейса в формат json.

Для создания противника так же воспользуемся интерфейсом. Интерфейс задаст общую функциональность для всех врагов, определит методы и свойства, которые должны реализовать все враги:

```
public interface IEnemy
    {
        string Name { get; }
        BigNumber BaseLife { get; }
        BigNumber Gold { get; };
        void TakeDamage(int damage);
    }
```

CEnemy будет базовым классом, реализующим общие свойства. Другие типы врагов будут наследоваться от него и расширять функциональность:

```
public abstract class CEnemy : IEnemy
    {
        public string Name { get; private set; }
        public BigNumber BaseLife { get; private set; }
        public BigNumber CurrentLife { get; private set; }
        public BigNumber Gold { get; private set; }

        protected CEnemyTemplate(string name, BigNumber baseLife, BigNumber gold)
        {
            Name = name;
            BaseLife = baseLife;
            CurrentLife = baseLife;
            Gold = gold;
        }


        public virtual void TakeDamage(BigNumber damage)
        {
            CurrentLife.Substruct(damage);
        }
    }
```

```
//Класс обычного противника не реализующего никаких особенностей
    public class CNormalEnemy : CEnemy { }
```

Для каждого из существующих уникальных классов требуется реализовать свое уникальное взаимодействие через переопределение метода TakeDamage – бронированный противник может вычитать получаемый урон своим показателем брони и т.д.

Реализуем «фабрику» для создания врагов используя принципом рефлексии – этот принцип позволяет создавать экземпляры классов динамически по строковому названию класса. Для этого реализуем класс фабрики:

```
public class EnemyFactory
    {
        public static IEnemy CreateEnemy(string typeName, params object[] args)
        {
            // Находим тип по названию
            var type = Assembly.GetExecutingAssembly()
                               .GetTypes()
                               .FirstOrDefault(t => t.Name == typeName);

            // Создаем объект с передачей параметров в конструктор
            return (IEnemy)Activator.CreateInstance(type, args);
        }
    }
```

Параметр **args** в данном случае означает параметры, которые требуется передать в конструктор класса:

```
//Простому противнику требуется задать имя, здоровье и золото
        IEnemy normalEnemy = EnemyFactory.CreateEnemy("CNormalEnemy", "Normal Enemy", 300,
50);
        //Бронированному противнику в свою очередь требуется задать еще и параметр брони
        IEnemy armoredEnemy = EnemyFactory.CreateEnemy("CArmoredEnemy", "Armored Enemy", 200,
25, 10);
```

Для реализации системы сохранения и загрузки можно воспользоваться реализованным ранее интерфейсом:

```
public interface ISaveList<T>
    {
        T Load(string path);
        void Save(T data, string path);
    }

    public class PlayerSaver : ISaveList<CPlayer>
    {
        public CPlayer Load(string path) { }
        public void Save(CPlayer player, string path){ }
    }
```