# Лабораторная работа № 3 Построение иерархии классов

**Цель лабораторной работы**

Познакомиться с основой объектного подхода в языке C#, созданием объектов, классов, и принципами объектно-ориентированного программирования.

**Теоретический материал**

*Классы и объекты*

**Объе́ктно-ориенти́рованное программирование (сокр. ООП)** – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих **объектов**, каждый из которых является экземпляром определённого **класса**, а классы образуют **иерархию наследования**.

Идеологически ООП – подход к программированию как к моделированию информационных **объектов**, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

Класс является основой для создания объектов. В классе определяются данные и код, который работает с этими данными. Объекты являются экземплярами класса. Методы и переменные, составляющие класс, называются членами класса. При определении класса объявляются данные, которые он содержит, и код, работающий с этими данными. Данные содержатся в переменных экземпляра, которые определены классом, а код содержится в методах. В С# определены несколько специфических разновидностей членов класса. Это — переменные экземпляра, статические переменные, константы, методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства.

Непосредственно инициализация переменных в объекте (переменных экземпляра) происходит в конструкторе. В классе могут быть определены несколько конструкторов.

**Синтаксис класса:**

*class имя\_класса{*

*тип\_доступа тип имя\_переменной1;*

*тип\_доступа тип имя\_переменной2;*

*…*

*тип\_доступа возвращаемый\_тип имя\_метода1(список\_параметров)*

*{*

*тело\_метода*

*}*

*}*

где тип\_доступа может быть public, private, protected, internal.

**Пример:**

// Класс Account инкапсулирует данные о банковском счете

public class Account

{

// Приватные поля для хранения данных счета

private decimal \_balance;

private string \_accountNumber;

// Конструктор для создания экземпляра счета с нулевым балансом

public Account()

{

\_balance = 0;

}

// Конструктор для создания экземпляра счета с указанным балансом

public Account(decimal initialBalance)

{

\_balance = initialBalance;

}

// Публичный метод для получения баланса счета

public decimal GetBalance()

{

return \_balance;

}

// Публичный метод для внесения средств на счет

public void Deposit(decimal amount)

{

ValidateAmount(amount); // Вызов приватного метода для проверки суммы

\_balance += amount;

}

// Публичный метод для снятия средств со счета

public void Withdraw(decimal amount)

{

ValidateAmount(amount); // Вызов приватного метода для проверки суммы

if (amount > \_balance)

{

throw new InvalidOperationException("Недостаточно средств на счете");

}

\_balance -= amount;

}

// Публичный метод для получения номера счета

public string GetAccountNumber()

{

return \_accountNumber;

}

// Публичный метод для установки номера счета

public void SetAccountNumber(string accountNumber)

{

\_accountNumber = accountNumber;

}

// Приватный метод для проверки суммы

private void ValidateAmount(decimal amount)

{

if (amount <= 0)

{

throw new ArgumentOutOfRangeException("Сумма должна быть положительной");

}

}

}

Создание обьекта:

*имя\_класса имя\_обьекта = new имя\_класса();*

При создании объекта класса происходит вызов соответствующего конструктора класса.

*Конструктор и деструктор*

Конструктор класса – метод для инициализации объекта при его создании. Он имеет то же имя, что и его класс. В конструкторах тип возвращаемого значения не указывается явно. Конструкторы используются для присваивания начальных значений переменным экземпляра, определенным классом, и для выполнения любых других процедур инициализации, необходимых для создания объекта.

Все классы имеют конструкторы независимо от того, определен он или нет. По умолчанию в С# предусмотрено наличие конструктора, который присваивает нулевые значения всем переменным экземпляра (для переменных обычных типов) и значения null (для переменных ссылочного типа). Но если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет.

*имя\_класса(список\_параметров) {тело\_конструктора}*

Деструктор – метод, вызывающийся автоматически при уничтожении объекта класса (непосредственно перед ―сборкой мусора‖). Деструктор не имеет параметров и возвращаемого значения.

*~имя\_класса() {тело\_деструктора}*

Объектно-ориентированное программирование (ООП) поддерживает *три основных принципа*:

- инкапсуляция,

- наследование,

- полиморфизм.

**Инкапсуляция**

*Инкапсуляция*– основной принцип ООП.

Состоит из двух аспектов:

- объединение данных и функций по работе с этими данными в единую сущность (описание класса),

- контроль доступа к элементам этой сущности (использование модификаторов доступа).

Инкапсуляция дает контроль над поведением объекта, но не более. Изменить атрибуты объекта возможно через предназначенный для этого интерфейс. Так, когда вы включаете телевизор, вы можете переключать каналы, регулировать громкость, но вы вряд ли знаете какие преобразования сигналов происходят в каждом блоке телевизора. Но при этом, если появится необходимость в ремонте телевизора, то для этого можно использовать сервисный центр.

Обычно считается правильным все данные в классе делать закрытыми извне, и если их необходимо изменять, то для этого предоставляют специальные методы. Члены класса с типом доступа *public* доступны везде за пределами данного класса, с типом доступа *protected* – внутри членов данного класса и производных, с типом доступа *private* – только для других членов данного класса. Тип доступа *internal* применяется для типов, доступных в пределах одной сборки.

**Свойство** в языке C# представляет собой специальный тип члена класса, который предоставляет простой способ доступа к частным полям класса. Свойства позволяют получать и устанавливать значения этих полей, как если бы они были публичными полями.

**Структура свойства:**

public тип\_возвращаемого\_значения Имя\_свойства { get; set; }

**Особенности свойств:**

* **get:** Аксессор для получения значения свойства.
* **set:** Аксессор для установки значения свойства.
* **тип\_возвращаемого\_значения:** Тип значения, возвращаемого свойством.
* **Имя\_свойства:** Имя свойства.

**Пример автосвойства:**

public class Person

{

public string FirstName { get; set; }

}

Также свойство можно записать в полном виде:

public class Person

{

private string \_firstName;

public string FirstName

{

get { return \_firstName; }

set { \_firstName = value; }

}

}

В этом примере свойство FirstName предоставляет доступ к частному полю \_firstName класса Person. Для получения значения свойства можно использовать следующий синтаксис:

string firstName = person.FirstName;

Для установки значения свойства можно использовать следующий синтаксис:

person.FirstName = "John";

**Преимущества использования свойств:**

* **Инкапсуляция:** Свойства позволяют инкапсулировать частные поля класса, обеспечивая контроль доступа к ним.
* **Валидация:** В аксессорах свойства можно выполнять валидацию значений, устанавливаемых для свойства.
* **Расширяемость:** Свойства могут быть переопределены в производных классах для предоставления дополнительной функциональности.
* **Простота использования:** Свойства упрощают доступ к данным класса, делая их доступными как обычные поля.

**Примечание:** Свойства не выделяют памяти, они просто предоставляют доступ к существующим полям.

**Пример** использования принципа инкапсуляции с помощью свойства:

public class Person

{

// Свойства с пользовательскими get и set

public string FirstName

{

get => \_firstName;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException("Имя не может быть пустой или содержать только пробелы.", nameof(value));

}

if (value.Length > 50)

{

throw new ArgumentException("Имя не может содержать более 50 символов.", nameof(value));

}

\_firstName = value;

}

}

public string LastName

{

get => \_lastName;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException("Фамилия не может быть пустой или содержать только пробелы.", nameof(value));

}

if (value.Length > 50)

{

throw new ArgumentException("Фамилия не может содержать более 50 символов.", nameof(value));

}

\_lastName = value;

}

}

private int \_age;

public int Age

{

get { return \_age; }

set { \_age = value < 0 ? 0 : value; }

}

// Конструктор

public Person(string firstName, string lastName, int age)

{

FirstName = firstName;

LastName = lastName;

Age = age;

}

// Метод вывода

public override string ToString()

{

return $"{FirstName} {LastName}, возраст: {Age}";

}

}

**Наследование**

Предположим, вы описываете автомобиль. А через некоторое время потребовалось описание абстракции «спортивный автомобиль». У вас есть два варианта: либо написать все описание с нуля (используя предыдущие описания), либо сказать, что спортивный автомобиль – это в первую очередь автомобиль, но с характерными параметрами.

В С# не поддерживается множественное наследование: класс может наследоваться только от одного класса. Однако, сюда добавлены интерфейсы, что позволяет создавать свой класс, реализовав в нем более чем один интерфейс.

**Пример** использования принципов наследования и инкапсуляции

1. Опишем класс следующим образом:

// Базовый класс Животное

public abstract class Animal

{

// Имя животного

public string Name

{

get => \_name;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException("Имя животного не может быть пустым или содержать только пробелы.", nameof(value));

}

if (value.Length > 50)

{

throw new ArgumentException("Имя животного не может содержать более 50 символов.", nameof(value));

}

\_name = value;

}

}

// Возраст животного

public int Age

{

get => \_age;

set

{

if (value < 0)

{

throw new ArgumentException("Возраст животного не может быть отрицательным.", nameof(value));

}

\_age = value;

}

}

// Абстрактный метод для воспроизведения звука животным

public abstract void MakeSound();

}

2. Опишем класс – наследник, у которого появляется дополнительная характеристика:

public class Cat : Animal

{

// Порода кошки

public string Breed

{

get => \_breed;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException("Порода кошки не может быть пустой или содержать только пробелы.", nameof(value));

}

if (value.Length > 50)

{

throw new ArgumentException("Порода кошки не может содержать более 50 символов.", nameof(value));

}

\_breed = value;

}

}

// Окрас кошки

public Color Color

{

get => \_color;

set

{

if (value == null)

{

throw new ArgumentNullException(nameof(value), "Окрас кошки не может быть null.");

}

\_color = value;

}

}

// Переопределение метода MakeSound для кошки

public override void MakeSound()

{

Console.WriteLine("Мяу!");

}

}

// Класс-наследник Собака

public class Dog : Animal

{

// Порода собаки

public string Breed

{

get => \_breed;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException("Порода собаки не может быть пустой или состоять только из пробелов.", nameof(value));

}

if (value.Length > 50)

{

throw new ArgumentException("Порода собаки не может содержать более 50 символов.", nameof(value));

}

\_breed = value;

}

}

public bool IsTrained { get; set; }

// Переопределение метода MakeSound для собаки

public override void MakeSound()

{

Console.WriteLine("Гав!");

}

}

// Класс-наследник Птица

public class Bird : Animal

{

// Размах крыльев птицы (в сантиметрах).

public int Wingspan

{

get => \_wingspan;

set

{

if (value <= 0)

{

throw new ArgumentException("Размах крыльев птицы должен быть положительным.", nameof(value));

}

\_wingspan = value;

}

}

// Переопределение метода MakeSound для птицы

public override void MakeSound()

{

Console.WriteLine("Чирик!");

}

}

3. Создаем экземпляр класса, приводим к типу базового класса и выводим доступную информацию:

// Тестовый класс

public class MainClass

{

public static void Main()

{

// Создание экземпляра класса Cat

Animal cat = new Cat

{

Name = "Мурзик",

Age = 5,

Breed = "Британец",

Color = Color.Black

};

// Создание экземпляра класса Dog

Animal dog = new Dog

{

Name = "Шарик",

Age = 7,

Breed = "Овчарка",

IsTrained = true

};

// Создание экземпляра класса Bird

Animal bird = new Bird

{

Name = "Кеша",

Age = 3,

Wingspan = 50

};

// Вызов метода MakeSound для каждого животного

cat.MakeSound(); // Выведет "Мяу!"

dog.MakeSound(); // Выведет "Гав!"

bird.MakeSound(); // Выведет "Чирик!"

}

}

**Полиморфизм**

*Полиморфизм*(от греческого слова *polymorphism,* означающего «много форм») – это качество, которое позволяет одному интерфейсу получать доступ к целому классу действий. Концепцию полиморфизма часто выражают такой фразой: «один интерфейс – много методов». Это означает, что для выполнения группы подобных действий можно разработать общий интерфейс. Полиморфизм позволяет понизить степень сложности программы, предоставляя программисту возможность использовать один и тот же интерфейс для задания *общего класса действий.* Конкретное (нужное в том или ином случае) действие (метод) выбирается компилятором. Программисту нет необходимости делать это вручную. Его задача – правильно использовать общий интерфейс.

Предположим, что мы создали класс музыканта, на основе которого построили два класса: гитарист и виолончелист.

Виолончелист

+ настраивать инструмент ()

Гитарист

+ настраивать инструмент ()

МУЗЫКАНТ

+ настраивать инструмент ()

Все музыканты умеют настраивать свои инструменты. Если мы хотим, чтобы целый оркестр настроитл свои инструменты, то нам проще обратиться ко всем музыкантам одновременно, а не отдельно ко всем гитаристам и ко всем виолончелистам. Полиморфизм позволяет вызвать именно ту функцию, которая необходима. То есть все музыканты начинают настраивать свои инструменты тем способом, который подходит для их инструментов, а не тем, которым умеет настраивать сам музыкант.

**Пример** использования принципа полиморфизма, виртуальных методов и работы с ними в производных классах.

В том случае, когда необходимо поддерживать концепцию полиморфизма при наследовании, выполняются два этапа:

1) определяется метод в базовом классе с ключевым словом *virtual* или *abstract***;**

2) переопределяется функция в производном классе с ключевым словом *override*(при перегрузке метода в производном классе) или *new*(при скрытии реализации метода базового класса).

Покажем работу на примере:

// Базовый класс Shape

public abstract class Shape

{

public abstract double Area();

}

// Производный класс Circle

public class Circle : Shape

{

private double \_radius;

public double Radius

{

get => \_radius;

set

{

if (value < 0)

{

throw new ArgumentOutOfRangeException("Radius", "Radius must be non-negative.");

}

\_radius = value;

}

}

public override double Area()

{

return Math.PI \* Radius \* Radius;

}

}

// Производный класс Rectangle

public class Rectangle : Shape

{

private double \_length;

public double Length

{

get => \_length;

set

{

if (value < 0)

{

throw new ArgumentOutOfRangeException("Length", "Length must be non-negative.");

}

\_length = value;

}

}

private double \_width;

public double Width

{

get => \_width;

set

{

if (value < 0)

{

throw new ArgumentOutOfRangeException("Width", "Width must be non-negative.");

}

\_width = value;

}

}

public override double Area()

{

return Length \* Width;

}

}

// Использование полиморфизма

List<Shape> shapes = new List<Shape>();

shapes.Add(new Circle { Radius = 5 });

shapes.Add(new Rectangle { Length = 10, Width = 5 });

foreach (var shape in shapes)

{

Console.WriteLine($"Площадь фигуры: {shape.Area()}");

}

В этом примере класс Shape является базовым классом, а классы Circle и Rectangle являются производными классами. Метод Area определен в базовом классе и переопределен в производных классах.

Когда мы создаем список фигур shapes и добавляем в него объекты Circle и Rectangle, мы можем вызвать метод Area для каждого объекта, и он будет вести себя по-разному в зависимости от типа объекта. Это демонстрирует полиморфизм в действии.

Рассмотрим пример со спец. словом virtual. Виртуальные методы позволяют производным классам переопределять реализацию методов базового класса.

// Базовый класс Animal

public class Animal

{

public double Weight { get; set; }

public virtual string MakeSound()

{

return "Неизвестный звук";

}

}

// Производный класс Dog

public class Dog : Animal

{

public override string MakeSound()

{

return "Гав-гав";

}

}

// Производный класс Cat

public class Cat : Animal

{

public override string MakeSound()

{

return "Мяу-мяу";

}

}

// Использование виртуальных методов

Animal animal = new Dog();

Console.WriteLine(animal.MakeSound()); // Выведет "Гав-гав"

animal = new Cat();

Console.WriteLine(animal.MakeSound()); // Выведет "Мяу-мяу"

В этом примере класс Animal является базовым классом, а классы Dog и Cat являются производными классами. Метод MakeSound определен как виртуальный в базовом классе, что позволяет производным классам переопределять его реализацию.

Когда мы создаем объект Dog и присваиваем его переменной animal, вызов метода MakeSound возвращает "Гав-гав", потому что метод MakeSound переопределен в классе Dog. Аналогично, когда мы создаем объект Cat и присваиваем его переменной animal, вызов метода MakeSound возвращает "Мяу-мяу", потому что метод MakeSound переопределен в классе Cat.

Это демонстрирует использование виртуальных методов для обеспечения полиморфного поведения в C#.

**Стандартные интерфейсы**

В C# для полного отделения структуры класса от его реализации используется механизм *интерфейсов*.

Интерфейс является расширением идеи абстрактных классов и методов. Синтаксис интерфейсов подобен синтаксису абстрактных классов. Объявление интерфейсов выполняется с помощью ключевого слова *interface*. При этом методы в интерфейсе не поддерживают реализацию. Членами интерфейса могут быть методы, свойства, индексаторы и события.

Интерфейс может реализовываться произвольным количеством классов. Один класс, в свою очередь, может реализовывать любое число интерфейсов. Каждый класс, включающий интерфейс, должен реализовывать его методы. В интерфейсе для методов неявным образом задается тип public. В этом случае также не допускается явный спецификатор доступа.

Синтаксис интерфейсов следующий:

[атрибуты] [модификаторы] interface Имя\_интерфейса [:список\_родительских\_интерфейсов]

{

обьявление\_свойств\_и\_методов

}

Пример:

interface Species

{

string Species();

void Feed();

}

class Cheetah : Animal, Species

{

private string ScientificName;

public string Species()

{

return ScientificName;

}

public void Feed()

{

Weight++;

}

}

Можно объявлять ссылочную переменную, имеющую интерфейсный тип. Подобная переменная может ссылаться на любой объект, который реализует ее интерфейс. При вызове метода объекта с помощью интерфейсной ссылки вызывается версия метода, реализуемого данным объектом.

Возможно наследование интерфейсов. В этом случае используется синтаксис, аналогичный наследованию классов. Если класс реализует интерфейс, который наследует другой интерфейс, должна обеспечиваться реализация для всех членов, определенных в составе цепи наследования интерфейсов.

*Стандартные интерфейсы*

В библиотеке .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс *IComparable* задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку. Реализация интерфейсов *IEnumerable* и *IEnumerator* дает возможность просматривать содержимое объекта с помощью конструкции foreach, а реализация интерфейса *ICloneable* – клонировать объекты.

Интерфейс IComparable содержит всего один метод CompareTo(), возвращающий результат сравнения двух объектов – текущего и переданного ему в качеству параметра:

interface IComparable

{

int CompareTo(object obj);

}

Метод возвращает:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра.
* Равен нулю. Значит, оба объекта равны.
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра.

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

interface IComparer

{

int Compare(object o1, object o2);

}

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше – то число меньше нуля; если оба объекта равны, возвращается ноль.

Для клонирования объектов удобно использовать интерфейс ICloneable. *Клонирование* – создание копии объекта. Копия объекта называется *клоном*. При присваивании одного экземпляра другому копируется ссылка, а не сам объект.

Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone(), который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется *поверхностным* *клонированием*.

Для создания полностью независимых объектов необходимо *глубокое* *копирование*, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей, и т.д. Алгоритм глубокого копирования требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Интерфейсы IEnumerable и IEnumerator позволяют перебирать объекты в цикле foreach:

foreach (var item in перечислимый\_объект)

{

}

Перебираемая коллекция должна реализовать интерфейс IEnumerable. Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс – перечислитель:

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

Интерфейс IEnumerator определяет функционал для перебора внутренних объектов:

public interface IEnumerator

{

bool MoveNext();//перемещение на одну позицию

//вперед в контейнере элементов

object Current { get; }//текущий элемент в контейнере

void Reset(); //перемещение в начало контейнера

}

* Метод MoveNext() перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.
* Свойство Current возвращает объект в последовательности, на который указывает указатель.
* Метод Reset() сбрасывает указатель позиции в начальное положение.

**Коллекции**

В C# *коллекция* – это совокупность объектов. Главное назначение коллекций в том, что они стандартизируют обработку групп объектов. Коллекции, в отличие от массивов, удобно применять, когда количество объектов заранее неизвестно. Все коллекции разработаны на основе набора четко определенных интерфейсов.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

*1. Необобщенные коллекции*

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". Необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Такие коллекции не типизированы, они оперируют данными типа object. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен System.Collections.

Классы необобщенных коллекций: ArrayList, Hashtable, Queue, SortedList, Stack.

*2. Специальные коллекции*

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

Классы специальных коллекций: CollectionsUtil, HybridDictionary, ListDictionary, NameValueCollection, OrderedDictionary, StringCollection, StringDictionary.

*3. Поразрядная коллекция*

Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

*4. Обобщенные коллекции*

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными, т.е. в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

Классы обобщенных коллекций:

1. Dictionary<Tkey, TValue>,
2. HashSet<T>,
3. LinkedList<T>,
4. List<T>,
5. Queue<T>,
6. SortedDictionary<TKey, TValue>,
7. SortedList<TKey, TValue>,
8. SortedSet<T>, Stack<T>.

*5. Параллельные коллекции*

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен System.Collections.Concurrent.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

Основополагающим для всех коллекций является понятие *перечислителя*, который поддерживается в интерфейсах IEnumerator и IEnumerable. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции – он перечисляет содержимое коллекции.

С перечислителем непосредственно связано другое средство, называемое *итератором*. Это средство упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле foreach.

Рассмотрим работу с коллекциями на примере класса *List<T>*. Класс *List*<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов. Основные методы класса:

* void Add(T item): добавление нового элемента в список;
* void AddRange(ICollection collection): добавление в список коллекции или массива;
* int BinarySearch(T item): бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции;
* int IndexOf(T item): возвращает индекс первого вхождения элемента в списке;
* void Insert(int index, T item): вставляет элемент item в списке на позицию index;
* bool Remove(T item): удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true;
* void RemoveAt(int index): удаление элемента по указанному индексу index;
* void Sort(): сортировка списка.

Пример создания простого списка, содержащего виды топлива:

static void Main(string[] args)

{

List<Wood> fuel = new List<Wood>(3); // (3) Capacity

fuel.Add(new Wood() { Name = "Щепа" });

fuel.Add(new Wood() { Name = "Пеллеты" });

foreach (Wood p in fuel)

{

Console.WriteLine(p.Name);

}

}

class Wood

{

public string Name { get; set; }

}

Здесь свойство *Capacity* позволяет получать и устанавливать емкость вызывающей коллекции. Емкость обозначает количество элементов, которые может содержать коллекция до ее вынужденного расширения. При начальном задании размерности исключаются издержки на выделение дополнительной памяти.

**Задание на лабораторную работу**

1. Разработать библиотеку классов (тип проекта).
2. В библиотеке классов создать классы в соответствии с вариантом, каждый класс в отдельном файле.
3. Реализовать стандартные интерфейсы: IEquatable<T>, IEnumerable<T>, ICloneable.
4. В этом же решении создать проект «Консольное приложение».
5. Подключить библиотеку классов к консольному приложению.
6. В консольном приложении разработать интерфейс, позволяющий взаимодействовать с реализованными классами:
7. создавать экземпляры классов;
8. вызывать методы классов.

**Примечание:**

1. Все указанные классы должны использоваться в программе.
2. Допускается дополнить предметную область дополнительными классами.
3. Допускается дополнить классы дополнительными полями и методами.
4. Допускается вносить незначительные изменения в схему и структуру классов (обсудить с преподавателем), если они значительно повышают соответствие программы предметной области.
5. Если что-то кажется непонятным – позвать преподавателя.

**Варианты заданий на лабораторную работу**

**1. Система управления библиотекой**

**Словесное описание**: Эта система предназначена для управления библиотечными ресурсами, включая книги и членов библиотеки. Она позволяет регистрировать новых пользователей, добавлять книги, выдавать и возвращать их, а также отслеживать доступность ресурсов.

**Классы и их поля:**

* **Book**
  + String Title — название книги.
  + String Author — автор книги.
  + String ISBN — уникальный идентификатор книги.
  + Boolean IsAvailable — статус доступности книги.
* **Member**
  + String Name — имя члена библиотеки.
  + String MemberID — уникальный идентификатор члена.
  + List<Book> LoanedBooks — список выданных книг.
* **Library**
  + List<Book> Books — коллекция книг в библиотеке.
  + List<Member> Members — список зарегистрированных членов библиотеки.

**Методы:**

* Library.AddBook(Book book)
* Library.RegisterMember(Member member)
* Library.LoanBook(Book book, Member member)
* Library.ReturnBook(Book book)
* Library.GetLoans()
* Library.SearchBook(String title)
* Library.GetAvailableBooks()
* Library.RemoveBook(Book book)
* Library.GetMembers()
* Library.GetBookDetails(Book book)

**2. Система управления заказами в ресторане**

**Словесное описание**: Эта система предназначена для управления меню ресторана, заказами клиентов и отслеживанием статуса заказов. Она позволяет добавлять блюда в меню, создавать заказы и управлять ими.

**Классы и их поля:**

* **Dish**
  + String Name — название блюда.
  + Double Price — цена блюда.
  + String Description — описание блюда.
* **Order**
  + String OrderID — уникальный идентификатор заказа.
  + List<Dish> Dishes — список блюд в заказе.
  + Double TotalAmount — общая сумма заказа.
  + String Status — статус заказа (например, "в процессе", "готов").
* **Restaurant**
  + List<Dish> Menu — меню ресторана.
  + List<Order> Orders — список всех заказов.

**Методы:**

* Restaurant.AddDish(Dish dish)
* Restaurant.CreateOrder(Customer customer)
* Order.AddDish(Dish dish)
* Order.CalculateTotal()
* Restaurant.GetOrderStatus(Order order)
* Restaurant.RemoveDish(Dish dish)
* Restaurant.GetMenu()
* Order.GetDishes()
* Customer.GetOrderHistory()
* Restaurant.UpdateDish(Dish dish)

**3. Управление проектами**

**Словесное описание**: Эта система предназначена для управления проектами и задачами в команде. Она позволяет добавлять задачи, назначать их сотрудникам, отслеживать статус выполнения и генерировать отчеты о прогрессе.

**Классы и их поля:**

* **Task**
  + String TaskID — уникальный идентификатор задачи.
  + String Description — описание задачи.
  + String Status — текущий статус задачи (например, "выполняется", "завершена").
  + Employee AssignedTo — сотрудник, которому назначена задача.
* **Project**
  + String ProjectID — уникальный идентификатор проекта.
  + String Name — название проекта.
  + List<Task> Tasks — список задач в проекте.
* **Employee**
  + String Name — имя сотрудника.
  + String EmployeeID — уникальный идентификатор сотрудника.
  + List<Task> AssignedTasks — список задач, назначенных сотруднику.

**Методы:**

* Project.AddTask(Task task)
* Task.AssignTo(Employee employee)
* Project.GetStatus()
* ProjectManager.GenerateReport(Project project)
* Project.RemoveTask(Task task)
* Project.GetTasks()
* Task.UpdateStatus(String status)
* Employee.GetAssignedTasks()
* ProjectManager.GetManagedProjects()
* Project.UpdateDescription(String description)

**4. Система управления учебным процессом**

**Словесное описание**: Эта система предназначена для управления учебным процессом, включая курсы, студентов и преподавателей. Она позволяет регистрировать студентов на курсы, отслеживать их успеваемость и управлять расписанием.

**Классы и их поля:**

* **Course**
  + String CourseID — уникальный идентификатор курса.
  + String Title — название курса.
  + List<Student> Students — список студентов, зачисленных на курс.
  + Instructor Instructor — преподаватель курса.
* **Student**
  + String Name — имя студента.
  + String StudentID — уникальный идентификатор студента.
  + List<Course> EnrolledCourses — список курсов, на которых зарегистрирован студент.
  + Dictionary<String, Double> Grades — оценки студента по курсам.
* **Instructor**
  + String Name — имя преподавателя.
  + String InstructorID — уникальный идентификатор преподавателя.
  + List<Course> CoursesTaught — список курсов, которые ведет преподаватель.

**Методы:**

* Course.AddStudent(Student student)
* Course.GetGrades()
* Instructor.AssignCourse(Course course)
* Student.GetCourses()
* Course.RemoveStudent(Student student)
* Course.GetInstructor()
* Instructor.GetCoursesTaught()
* Enrollment.GetEnrollmentDate()
* Student.UpdateGrades(String courseID, Double grade)
* Course.GetEnrolledStudents()

**5. Система учета финансов**

**Словесное описание**: Эта система предназначена для учета финансовых операций, включая доходы и расходы. Она позволяет отслеживать транзакции, управлять бюджетами и генерировать финансовые отчеты.

**Классы и их поля:**

* **Transaction**
  + string TransactionID — уникальный идентификатор транзакции.
  + string Description — описание транзакции.
  + decimal Amount — сумма транзакции (может быть положительной для доходов и отрицательной для расходов).
  + DateTime Date — дата транзакции.
* **Account**
  + string AccountID — уникальный идентификатор счета.
  + string AccountName — название счета.
  + List<Transaction> Transactions — список всех транзакций, связанных с этим счетом.
  + decimal Balance — текущий баланс счета (доступен только для чтения).
* **FinanceManager**
  + List<Account> Accounts — список всех счетов.

**Методы:**

* Account.ToString(Transaction transaction)
* Account.AddTransaction(Transaction transaction) — метод для добавления новой транзакции и обновления баланса.
* Account.GetTransactions() — метод для получения списка всех транзакций.
* Account.GetBalance() — метод для получения текущего баланса.
* FinanceManager.AddAccount(Account account) — метод для добавления нового счета в систему.
* FinanceManager.GetAccount(string accountID) — метод для получения счета по его уникальному идентификатору.
* FinanceManager.PrintAllAccounts() — метод для вывода информации о всех счетах и их балансах.

**6. Система бронирования отелей**

**Словесное описание**: Эта система предназначена для управления бронированием гостиничных номеров. Она позволяет добавлять номера, проверять их доступность, оформлять и отменять бронирования.

**Классы и их поля:**

* **Room**
  + String RoomNumber — номер комнаты.
  + String Type — тип номера (например, "одноместный", "двухместный").
  + Double Price — цена номера.
  + Boolean IsAvailable — статус доступности номера.
* **Reservation**
  + String ReservationID — уникальный идентификатор бронирования.
  + Customer Customer — клиент, сделавший бронирование.
  + Room Room — номер, который был забронирован.
  + DateTime CheckInDate — дата заезда.
  + DateTime CheckOutDate — дата выезда.
* **Hotel**
  + String HotelName — название отеля.
  + List<Room> Rooms — список номеров в отеле.
  + List<Reservation> Reservations — список всех бронирований.

**Методы:**

* Hotel.AddRoom(Room room)
* Hotel.CheckAvailability(Room room)
* Hotel.MakeReservation(Customer customer, Room room)
* Hotel.CancelReservation(Reservation reservation)
* Hotel.GetRooms()
* Room.GetRoomDetails()
* Hotel.GetReservations()
* Customer.GetReservationHistory()
* Hotel.UpdateRoom(Room room)
* Hotel.GetAvailableRooms()

**7. Система управления мультимедиа**

**Словесное описание**: Эта система предназначена для управления мультимедийными материалами, включая фильмы, музыку и книги. Она позволяет добавлять медиафайлы, отслеживать их доступность и выдавать пользователям.

**Классы и их поля:**

* **Media**
  + String Title — название мультимедийного материала.
  + String Type — тип материала (например, "фильм", "книга").
  + String Genre — жанр.
  + Boolean IsAvailable — статус доступности.
* **Library**
  + List<Media> MediaCollection — коллекция мультимедийных материалов.
  + List<Member> Members — список зарегистрированных членов.

**Методы:**

* Library.AddMedia(Media media)
* Library.LoanMedia(Media media)
* Library.ReturnMedia(Media media)
* Library.SearchMedia(String title)
* Library.GetMediaCollection()
* Media.GetMediaDetails()
* Library.RemoveMedia(Media media)
* Library.GetAvailableMedia()
* Library.UpdateMedia(Media media)
* Library.GetMediaByType(String type)

**8. Система управления спортивной командой**

**Словесное описание**: Эта система предназначена для управления спортивной командой, включая игроков, матчи и результаты. Она позволяет добавлять игроков, планировать матчи и отслеживать результаты.

**Классы и их поля:**

* **Player**
  + String Name — имя игрока.
  + String PlayerID — уникальный идентификатор игрока.
  + String Position — позиция игрока.
  + Dictionary<String, int> Stats — статистика игрока.
* **Team**
  + String TeamName — название команды.
  + List<Player> Players — список игроков команды.
  + List<Match> Matches — список матчей команды.
* **Match**
  + String MatchID — уникальный идентификатор матча.
  + Team FirstTeam — первая команда.
  + Team SecondTeam — вторая команда.
  + DateTime Date — дата матча.
  + String Result — результат матча.

**Методы:**

* Team.AddPlayer(Player player)
* Team.ScheduleMatch(Match match)
* Team.GetStandings()
* Player.UpdateStats(String statType, int value)
* Team.RemovePlayer(Player player)
* Match.GetMatchDetails()
* Team.GetPlayers()
* League.GetTeams()
* Team.GetMatches()
* Player.GetPlayerStats()

**9. Система управления событиями**

**Описание**: Эта система помогает организовывать и управлять различными событиями, такими как конференции, выставки и семинары.

**Классы и их поля:**

* **Event**
  + String EventID: Уникальный идентификатор события.
  + String Title: Название события.
  + DateTime Date: Дата и время события.
  + String Location: Место проведения события.
  + String Description: Описание события.
* **Participant**
  + String ParticipantID: Уникальный идентификатор участника.
  + String Name: Имя участника.
  + String Email: Электронная почта участника.
  + String PhoneNumber: Номер телефона участника.
* **EventManager**
  + List<Event> Events: Список всех событий.
  + List<Participant> Participants: Список всех участников.
  + String OrganizerName: Имя организатора.
  + String ContactInfo: Контактная информация организатора.

**Методы:**

* EventManager.AddEvent(Event event)
* EventManager.RegisterParticipant(Event event, Participant participant)
* EventManager.GetEventDetails(Event event)
* EventManager.GetParticipants(Event event)
* EventManager.RemoveParticipant(Event event, Participant participant)

**10. Система управления складом**

**Описание**: Эта система предназначена для управления запасами на складе. Она позволяет отслеживать поступления и отгрузки товаров, управлять запасами и генерировать отчеты о состоянии склада.

**Классы и их поля:**

* **Product**
  + String ProductID: Уникальный идентификатор товара.
  + String Name: Название товара.
  + Int Quantity: Количество на складе.
  + Double Price: Цена товара.
  + String Category: Категория товара.
* **Warehouse**
  + String WarehouseID: Уникальный идентификатор склада.
  + String Location: Местоположение склада.
  + List<Product> Products: Список всех товаров на складе.
  + Double TotalValue: Общая стоимость товаров на складе.
* **InventoryManager**
  + List<Warehouse> Warehouses: Список всех складов.
  + List<Product> AllProducts: Список всех товаров.
  + String ManagerName: Имя менеджера склада.
  + String ContactInfo: Контактная информация менеджера.

**Методы:**

* InventoryManager.AddProduct(Product product)
* InventoryManager.UpdateProductQuantity(Product product, Int quantity)
* InventoryManager.RemoveProduct(Product product)
* InventoryManager.GenerateInventoryReport()
* InventoryManager.GetWarehouseDetails(String warehouseID)

**11. Система управления контактами**

**Описание**: Эта система позволяет пользователям хранить и управлять контактной информацией. Она предоставляет возможность добавления, редактирования и удаления контактов, а также поиска по ним.

**Классы и их поля:**

* **Contact**
  + String ContactID: Уникальный идентификатор контакта.
  + String Name: Имя контакта.
  + String PhoneNumber: Номер телефона.
  + String Email: Электронная почта.
  + String Address: Адрес контакта.
* **ContactGroup**
  + String GroupID: Уникальный идентификатор группы.
  + String GroupName: Название группы.
  + List<Contact> Members: Список контактов в группе.
  + String Description: Описание группы.
* **ContactManager**
  + List<Contact> Contacts: Список всех контактов.
  + List<ContactGroup> Groups: Список всех групп.
  + String ManagerName: Имя менеджера контактов.
  + String ContactInfo: Контактная информация менеджера.

**Методы:**

* ContactManager.AddContact(Contact contact)
* ContactManager.EditContact(Contact contact)
* ContactManager.RemoveContact(Contact contact)
* ContactManager.CreateGroup(ContactGroup group)
* ContactManager.GetAllContacts()

**12. Система управления задачами для личного использования**

**Описание**: Эта система помогает пользователям управлять своими личными задачами и делами. Она позволяет добавлять, редактировать и отслеживать выполнение задач.

**Классы и их поля:**

* **Task**
  + String TaskID: Уникальный идентификатор задачи.
  + String Description: Описание задачи.
  + Boolean IsCompleted: Статус выполнения задачи.
  + DateTime DueDate: Срок выполнения задачи.
  + String Priority: Приоритет задачи (например, "высокий", "средний", "низкий").
* **TaskList**
  + String ListID: Уникальный идентификатор списка задач.
  + String Title: Название списка задач.
  + List<Task> Tasks: Список задач.
  + DateTime CreatedDate: Дата создания списка.
* **TaskManager**
  + List<TaskList> TaskLists: Список всех списков задач.
  + String ManagerName: Имя менеджера задач.
  + String ContactInfo: Контактная информация менеджера.
  + Int TotalTasks: Общее количество задач.

**Методы:**

* TaskManager.AddTask(Task task, TaskList list)
* TaskManager.EditTask(Task task)
* TaskManager.RemoveTask(Task task)
* TaskManager.GetPendingTasks()
* TaskManager.GetCompletedTasks()

**13. Система управления подписками**

**Описание**: Эта система позволяет управлять подписками на различные услуги или контент. Пользователи могут подписываться, отменять подписки и отслеживать свои подписки.

**Классы и их поля:**

* **Subscription**
  + String SubscriptionID: Уникальный идентификатор подписки.
  + String ServiceName: Название услуги.
  + DateTime StartDate: Дата начала подписки.
  + DateTime EndDate: Дата окончания подписки.
  + Double Price: Цена подписки.
* **User**
  + String UserID: Уникальный идентификатор пользователя.
  + String Name: Имя пользователя.
  + String Email: Электронная почта пользователя.
  + List<Subscription> Subscriptions: Список подписок пользователя.
* **SubscriptionManager**
  + List<User> Users: Список всех пользователей.
  + List<Subscription> AllSubscriptions: Список всех подписок.
  + String ManagerName: Имя менеджера подписок.
  + String ContactInfo: Контактная информация менеджера.

**Методы:**

* SubscriptionManager.AddSubscription(User user, Subscription subscription)
* SubscriptionManager.CancelSubscription(User user, Subscription subscription)
* SubscriptionManager.GetActiveSubscriptions(User user)
* SubscriptionManager.GetExpiredSubscriptions(User user)
* SubscriptionManager.GetSubscriptionDetails(Subscription subscription)

**14. Система управления резюме**

**Описание**: Эта система позволяет пользователям создавать и управлять своими резюме. Она предоставляет возможность добавления различных секций, редактирования и экспорта резюме в разных форматах.

**Классы и их поля:**

* **Resume**
  + String ResumeID: Уникальный идентификатор резюме.
  + String Name: Имя соискателя.
  + String ContactInfo: Контактная информация.
  + List<String> WorkExperience: Список опыта работы.
  + List<String> Education: Список образования.
* **Section**
  + String SectionID: Уникальный идентификатор секции.
  + String Title: Название секции (например, "Опыт работы").
  + List<String> Content: Содержимое секции.
  + Int Order: Порядок отображения секции.
* **ResumeManager**
  + List<Resume> Resumes: Список всех резюме.
  + String ManagerName: Имя менеджера резюме.
  + String ContactInfo: Контактная информация менеджера.
  + Int TotalResumes: Общее количество резюме.

**Методы:**

* ResumeManager.CreateResume(Resume resume)
* ResumeManager.EditResume(Resume resume)
* ResumeManager.DeleteResume(Resume resume)
* ResumeManager.ExportResume(Resume resume, String format)
* ResumeManager.GetAllResumes()

**15. Система управления опросами**

**Описание**: Эта система позволяет создавать и управлять опросами. Пользователи могут создавать опросы, добавлять вопросы и анализировать результаты.

**Классы и их поля:**

* **Survey**
  + String SurveyID: Уникальный идентификатор опроса.
  + String Title: Название опроса.
  + DateTime StartDate: Дата начала опроса.
  + DateTime EndDate: Дата окончания опроса.
  + List<Question> Questions: Список вопросов в опросе.
* **Question**
  + String QuestionID: Уникальный идентификатор вопроса.
  + String Text: Текст вопроса.
  + List<String> Options: Возможные варианты ответов.
  + String QuestionType: Тип вопроса (например, "один выбор", "несколько выборов").
* **Response**
  + String ResponseID: Уникальный идентификатор ответа.
  + String SurveyID: Идентификатор опроса.
  + Dictionary<String, String> Answers: Ответы на вопросы.
  + DateTime SubmissionDate: Дата подачи ответа.

**Методы:**

* SurveyManager.CreateSurvey(Survey survey)
* SurveyManager.AddQuestion(Survey survey, Question question)
* SurveyManager.GetSurveyResults(Survey survey)
* SurveyManager.DeleteSurvey(Survey survey)
* SurveyManager.GetAllSurveys()

**16. Система управления благотворительными проектами**

**Описание**: Эта система предназначена для управления благотворительными проектами, сборами средств и волонтерами.

**Классы и их поля:**

* **CharityProject**
  + String ProjectID: Уникальный идентификатор проекта.
  + String Title: Название проекта.
  + Double GoalAmount: Целевая сумма сбора средств.
  + Double CollectedAmount: Собранная сумма.
  + String Description: Описание проекта.
* **Donor**
  + String DonorID: Уникальный идентификатор донора.
  + String Name: Имя донора.
  + Double AmountDonated: Сумма пожертвования.
  + DateTime DonationDate: Дата пожертвования.
  + String ContactInfo: Контактная информация донора.
* **Volunteer**
  + String VolunteerID: Уникальный идентификатор волонтера.
  + String Name: Имя волонтера.
  + List<CharityProject> Projects: Список благотворительных проектов, в которых участвует волонтер.
  + Int HoursWorked: Количество часов, отработанных волонтером.
  + String ContactInfo: Контактная информация волонтера.

**Методы:**

* CharityProject.AddDonor(Donor donor)
* CharityProject.UpdateCollectedAmount(Double amount)
* Volunteer.JoinProject(CharityProject project)
* Volunteer.LeaveProject(CharityProject project)
* CharityProject.GetProjectDetails(String projectID)

**17. Система управления фитнес-программами**

**Описание**: Эта система позволяет пользователям управлять своими фитнес-программами, отслеживать тренировки и прогресс.

**Классы и их поля:**

* **Workout**
  + String WorkoutID: Уникальный идентификатор тренировки.
  + String Name: Название тренировки.
  + List<Exercise> Exercises: Список упражнений.
  + DateTime Date: Дата тренировки.
  + String Duration: Продолжительность тренировки.
* **Exercise**
  + String ExerciseID: Уникальный идентификатор упражнения.
  + String Name: Название упражнения.
  + Int Sets: Количество подходов.
  + Int Reps: Количество повторений.
  + String Notes: Примечания к упражнению.
* **FitnessProgram**
  + String ProgramID: Уникальный идентификатор программы.
  + String UserID: Идентификатор пользователя.
  + List<Workout> Workouts: Список тренировок в программе.
  + String Goals: Цели программы (например, "похудение", "набор массы").
  + DateTime StartDate: Дата начала программы.

**Методы:**

* FitnessProgram.AddWorkout(Workout workout)
* Workout.AddExercise(Exercise exercise)
* FitnessProgram.GetProgress()
* FitnessProgram.RemoveWorkout(Workout workout)
* FitnessProgram.GetWorkoutDetails(Workout workout)

**19. Система управления проектами**

**Описание**: Эта система помогает командам управлять проектами, отслеживать прогресс и координировать задачи.

**Классы и их поля:**

* **Project**
  + String ProjectID: Уникальный идентификатор проекта.
  + String Title: Название проекта.
  + DateTime StartDate: Дата начала проекта.
  + DateTime EndDate: Дата окончания проекта.
  + String Status: Статус проекта (например, "в процессе", "завершен").
* **Task**
  + String TaskID: Уникальный идентификатор задачи.
  + String Description: Описание задачи.
  + String AssignedTo: Имя ответственного за задачу.
  + DateTime DueDate: Дата выполнения задачи.
  + String Priority: Приоритет задачи.
* **TeamMember**
  + String MemberID: Уникальный идентификатор участника команды.
  + String Name: Имя участника.
  + String Role: Роль в проекте (например, "менеджер", "разработчик").
  + String ContactInfo: Контактная информация.
  + List<Task> AssignedTasks: Список задач, назначенных участнику.

**Методы:**

* Project.AddTask(Task task)
* Project.UpdateStatus(String status)
* TeamMember.AssignTask(Task task)
* Project.GetProjectDetails()
* TeamMember.GetAssignedTasks()

**20. Система управления складом**

**Описание**: Эта система помогает управлять запасами на складе, отслеживать поступления и отгрузки товаров, а также управлять поставщиками.

**Классы и их поля:**

* **Product**
  + String ProductID: Уникальный идентификатор товара.
  + String Name: Название товара.
  + String Description: Описание товара.
  + Double Price: Цена товара.
  + Int Quantity: Количество товара на складе.
* **Supplier**
  + String SupplierID: Уникальный идентификатор поставщика.
  + String Name: Название компании-поставщика.
  + String ContactInfo: Контактная информация поставщика.
  + List<Product> SuppliedProducts: Список товаров, предоставляемых поставщиком.
* **InventoryTransaction**
  + String TransactionID: Уникальный идентификатор транзакции.
  + String ProductID: Идентификатор товара.
  + Int Quantity: Количество, добавляемое или убираемое.
  + DateTime TransactionDate: Дата транзакции.
  + String Type: Тип транзакции ("поступление" или "отгрузка").

**Методы:**

* Product.AddStock(Int quantity)
* Product.RemoveStock(Int quantity)
* Supplier.AddProduct(Product product)
* InventoryTransaction.RecordTransaction(Product product, Int quantity, String type)
* Product.GetProductDetails()

**21. Система управления ресторанами**

**Описание**: Эта система помогает ресторанам управлять заказами, меню и клиентами.

**Классы и их поля:**

* **MenuItem**
  + String ItemID: Уникальный идентификатор блюда.
  + String Name: Название блюда.
  + Double Price: Цена блюда.
  + String Description: Описание блюда.
  + Boolean IsAvailable: Статус доступности блюда.
* **Order**
  + String OrderID: Уникальный идентификатор заказа.
  + List<MenuItem> Items: Список заказанных блюд.
  + DateTime OrderDate: Дата заказа.
  + String Status: Статус заказа (например, "в процессе", "завершен").
* **Customer**
  + String CustomerID: Уникальный идентификатор клиента.
  + String Name: Имя клиента.
  + String ContactInfo: Контактная информация.
  + List<Order> Orders: Список заказов клиента.

**Методы:**

* Order.AddMenuItem(MenuItem item)
* Order.UpdateStatus(String status)
* MenuItem.UpdatePrice(Double price)
* Customer.GetOrderHistory()
* MenuItem.GetDetails()

**22. Система управления мероприятиями**

**Описание**: Эта система предназначена для планирования и управления различными мероприятиями, такими как конференции, выставки и семинары.

**Классы и их поля:**

* **Event**
  + String EventID: Уникальный идентификатор мероприятия.
  + String Title: Название мероприятия.
  + String Description: Описание мероприятия.
  + DateTime Date: Дата и время проведения мероприятия.
  + String Location: Место проведения мероприятия.
* **Attendee**
  + String AttendeeID: Уникальный идентификатор участника.
  + String Name: Имя участника.
  + String Email: Электронная почта участника.
  + List<Event> RegisteredEvents: Список мероприятий, на которые зарегистрирован участник.
* **Sponsor**
  + String SponsorID: Уникальный идентификатор спонсора.
  + String Name: Название компании-спонсора.
  + String ContactInfo: Контактная информация спонсора.
  + List<Event> SponsoredEvents: Список мероприятий, которые спонсирует компания.

**Методы:**

* Event.RegisterAttendee(Attendee attendee)
* Event.UpdateDetails(String title, String description, DateTime date, String location)
* Sponsor.AddEvent(Event event)
* Attendee.GetRegisteredEvents()
* Event.GetEventDetails()

**23. Система управления финансами**

**Описание**: Эта система помогает пользователям отслеживать свои доходы, расходы и составлять бюджеты.

**Классы и их поля:**

* **Transaction**
  + String TransactionID: Уникальный идентификатор транзакции.
  + String Description: Описание транзакции.
  + Double Amount: Сумма транзакции.
  + DateTime Date: Дата транзакции.
  + String Category: Категория (например, "доход", "расход").
* **Budget**
  + String BudgetID: Уникальный идентификатор бюджета.
  + String Name: Название бюджета.
  + Double TotalAmount: Общая сумма бюджета.
  + Double SpentAmount: Сумма, потраченная из бюджета.
  + List<Transaction> Transactions: Список транзакций в рамках бюджета.
* \*\*User \*\*
  + String UserID: Уникальный идентификатор пользователя.
  + String Name: Имя пользователя.
  + String Email: Электронная почта пользователя.
  + List<Budget> Budgets: Список бюджетов пользователя.

**Методы:**

* User .AddTransaction(Transaction transaction)
* Budget.UpdateSpentAmount(Double amount)
* User .GetTransactionHistory()
* Budget.GetBudgetDetails()
* User .GetBudgets()

**24. Система управления путешествиями**

**Описание**: Эта система помогает пользователям планировать и организовывать свои путешествия.

**Классы и их поля:**

* **Trip**
  + String TripID: Уникальный идентификатор путешествия.
  + String Destination: Место назначения.
  + DateTime StartDate: Дата начала путешествия.
  + DateTime EndDate: Дата окончания путешествия.
  + List<Activity> Activities: Список запланированных мероприятий.
* **Activity**
  + String ActivityID: Уникальный идентификатор мероприятия.
  + String Name: Название мероприятия.
  + String Description: Описание мероприятия.
  + DateTime Date: Дата мероприятия.
  + String Location: Место проведения мероприятия.
* **Traveler**
  + String TravelerID: Уникальный идентификатор путешественника.
  + String Name: Имя путешественника.
  + String ContactInfo: Контактная информация.
  + List<Trip> Trips: Список поездок путешественника.

**Методы:**

* Trip.AddActivity(Activity activity)
* Traveler.GetTripHistory()
* Trip.GetTripDetails()
* Activity.UpdateDetails(String description, DateTime date)
* Traveler.RemoveTrip(Trip trip)

**25. Система управления медицинскими записями**

**Описание**: Эта система позволяет медицинским учреждениям управлять записями пациентов, назначениями и медицинской историей.

**Классы и их поля:**

* **Patient**
  + String PatientID: Уникальный идентификатор пациента.
  + String Name: Имя пациента.
  + String ContactInfo: Контактная информация.
  + List<MedicalRecord> MedicalRecords: Список медицинских записей.
* **Appointment**
  + String AppointmentID: Уникальный идентификатор назначения.
  + String PatientID: Идентификатор пациента.
  + DateTime AppointmentDate: Дата и время назначения.
  + String Doctor: Имя врача.
  + String Reason: Причина назначения.
* **MedicalRecord**
  + String RecordID: Уникальный идентификатор записи.
  + String PatientID: Идентификатор пациента.
  + String Diagnosis: Диагноз.
  + String Treatment: Лечение.
  + DateTime RecordDate: Дата записи.

**Методы:**

* Patient.AddMedicalRecord(MedicalRecord record)
* Patient.GetAppointments()
* Appointment.UpdateDetails(DateTime date, String doctor)
* MedicalRecord.GetRecordDetails()
* Patient.RemoveAppointment(Appointment appointment)

**26. Система управления событиями для некоммерческих организаций**

**Описание**: Эта система помогает некоммерческим организациям управлять мероприятиями, сборами средств и волонтерами.

**Классы и их поля:**

* **Event**
  + String EventID: Уникальный идентификатор мероприятия.
  + String Title: Название мероприятия.
  + DateTime Date: Дата мероприятия.
  + String Location: Место проведения мероприятия.
  + Double FundraisingGoal: Целевая сумма сбора средств.
* **Volunteer**
  + String VolunteerID: Уникальный идентификатор волонтера.
  + String Name: Имя волонтера.
  + String ContactInfo: Контактная информация.
  + List<Event> Events: Список мероприятий, в которых участвует волонтер.
* **Donation**
  + String DonationID: Уникальный идентификатор пожертвования.
  + String DonorName: Имя донора.
  + Double Amount: Сумма пожертвования.
  + DateTime DonationDate: Дата пожертвования.
  + String EventID: Идентификатор мероприятия, на которое сделано пожертвование.

**Методы:**

* Event.AddVolunteer(Volunteer volunteer)
* Event.UpdateFundraisingGoal(Double amount)
* Donation.RecordDonation(String eventID, Double amount)
* Volunteer.GetEventParticipation()
* Event.GetEventDetails()

**Пример разработки приложения**

**Система управления заказами в интернет-магазине**

**Описание**: Эта система предназначена для управления заказами в интернет-магазине, включая управление товарами, клиентами и статусами заказов.

**Шаг 1. Создание проекта**

1. Откройте **Visual Studio**.
2. Нажмите **"Создать новый проект"**.
3. Выберите **"Консольное приложение"**.
4. Назовите проект (например, OnlineShopManagement) и нажмите **"Создать"**.

**Шаг 2. Создание перечисления**OrderStatus

В файле Program.cs (или создайте новый файл OrderStatus.cs) добавьте:

// Enum для статуса заказа

public enum OrderStatus

{

Processing,

Shipped,

Delivered,

Canceled

}

**Зачем это нужно?** Это список возможных статусов заказа, которые будут использоваться в классе Order.

**Шаг 3. Создание класса**Product**(Товар)**

1. Создайте новый файл Product.cs (или добавьте в Program.cs).
2. Добавьте свойства:
   * ProductID (уникальный код товара)
   * Name (название)
   * Price (цена)
   * StockQuantity (количество на складе)
3. Реализуйте методы:
   * AddStock() – увеличивает количество товара.
   * RemoveStock() – уменьшает количество (если товара достаточно).
   * CalculateTotalPrice() – вычисляет стоимость для указанного количества.
   * GetProductSummary() – возвращает строку с информацией о товаре.
4. **Проверка:**

Убедитесь, что класс реализует интерфейсы ICloneable и IEquatable<Product>.

// Класс для представления продукта

public class Product : ICloneable, IEquatable<Product>

{

public string ProductID { get; private set; }

public string Name { get; private set; }

public double Price { get; private set; }

public int StockQuantity { get; private set; }

public Product(string productId, string name, double price, int stockQuantity)

{

ProductID = productId;

Name = name;

Price = price;

StockQuantity = stockQuantity;

}

public void AddStock(int quantity)

{

StockQuantity += quantity;

}

public bool RemoveStock(int quantity)

{

if (StockQuantity >= quantity)

{

StockQuantity -= quantity;

return true;

}

else

{

return false;

}

}

public double CalculateTotalPrice(int quantity)

{

return Price \* quantity;

}

public string GetProductSummary()

{

return $"{Name} (ID: {ProductID}) - Цена: {Price}, Остаток: {StockQuantity}";

}

public override string ToString()

{

return GetProductSummary();

}

public object Clone()

{

return MemberwiseClone();

}

public bool Equals(Product? other)

{

if (other == null)

return false;

return other.ProductID == ProductID;

}

public override bool Equals(object obj)

{

return Equals(obj as Product);

}

public override int GetHashCode()

{

return ProductID.GetHashCode();

}

}

**Шаг 4. Создание класса**Customer**(Клиент)**

1. Создайте файл Customer.cs.
2. Добавьте свойства:
   * CustomerID (уникальный код клиента)
   * Name (имя)
   * Email (email, с проверкой формата)
   * Address (адрес, с проверкой)
   * Orders (список заказов клиента)
3. Реализуйте методы:
   * SetEmail() – проверяет, содержит ли email символ @.
   * SetAddress() – проверяет, что адрес не пустой.
   * AddOrder() – добавляет заказ в список.
   * HasOrders() – проверяет, есть ли заказы у клиента.
4. **Проверка:**

Убедитесь, что класс реализует ICloneable и IEquatable<Customer>.

// Класс для представления клиента

public class Customer : ICloneable, IEquatable<Customer>

{

public string CustomerID { get; private set; }

public string Name { get; private set; }

public IReadOnlyList<Order> Orders => orders.AsReadOnly();

private string email;

private string address;

private List<Order> orders;

public Customer(string customerId, string name, string email, string address)

{

CustomerID = customerId;

Name = name;

SetEmail(email);

SetAddress(address);

orders = new List<Order>();

}

public void SetEmail(string newEmail)

{

if (IsValidEmail(newEmail))

{

email = newEmail;

}

else

{

throw new ArgumentException("Неверный формат электронной почты.");

}

}

public void SetAddress(string newAddress)

{

if (IsValidAddress(newAddress))

{

address = newAddress;

}

else

{

throw new ArgumentException("Неверный формат адреса.");

}

}

public string GetCustomerInfo()

{

return $"Клиент: {Name} (ID: {CustomerID}), Email: {email}, Адрес: {address}";

}

public override string ToString()

{

return GetCustomerInfo();

}

public bool HasOrders()

{

return orders.Count > 0; // Проверка на наличие заказов

}

public void AddOrder(Order order)

{

orders.Add(order);

}

public void AddOrders(IEnumerable<Order> orders)

{

foreach (var order in orders)

{

AddOrder(order);

}

}

private bool IsValidEmail(string email)

{

// Проверка на наличие символа "@" и точки "."

return email.Contains("@") && email.IndexOf('.') > email.IndexOf('@') + 1;

}

private bool IsValidAddress(string address)

{

// Проверка, что адрес не пустой и содержит хотя бы 5 символов

return !string.IsNullOrWhiteSpace(address) && address.Length >= 5;

}

public object Clone()

{

var customer = new Customer(CustomerID, Name, email, address);

var orders = new List<Order>();

foreach (var order in Orders)

{

customer.AddOrder((Order)order.Clone());

}

return customer;

}

public bool Equals(Customer? other)

{

if (other == null)

return false;

return other.CustomerID == CustomerID;

}

public override bool Equals(object obj)

{

return Equals(obj as Customer);

}

public override int GetHashCode()

{

return CustomerID.GetHashCode();

}

}

**Шаг 5. Создание класса**Order**(Заказ)**

1. Создайте файл Order.cs.
2. Добавьте свойства:
   * OrderID (уникальный код заказа)
   * Customer (клиент, сделавший заказ)
   * Products (список товаров в заказе)
   * OrderDate (дата создания)
   * Status (статус из OrderStatus)
3. Реализуйте методы:
   * AddProduct() – добавляет товар в заказ (уменьшает остаток на складе).
   * UpdateStatus() – меняет статус заказа.
   * CalculateTotalOrderValue() – считает общую сумму заказа.
   * GetOrderSummary() – возвращает строку с информацией о заказе.
4. **Проверка:**

Убедитесь, что класс реализует ICloneable, IEquatable<Order> и IEnumerable<Product>.

// Класс для представления заказа

public class Order : ICloneable, IEquatable<Order>, IEnumerable<Product>

{

public string OrderID { get; private set; }

public Customer Customer { get; private set; }

public IReadOnlyList<Product> Products => products.AsReadOnly();

public DateTime OrderDate { get; private set; }

public OrderStatus Status { get; private set; }

private List<Product> products;

public Order(string orderId, Customer customer)

{

OrderID = orderId;

Customer = customer;

products = new List<Product>();

OrderDate = DateTime.Now;

Status = OrderStatus.Processing;

}

public bool AddProduct(Product product)

{

if (product.RemoveStock(1)) // Уменьшаем количе-ство товара на складе

{

products.Add(product);

return true;

}

return false;

}

public void UpdateStatus(OrderStatus status)

{

Status = status;

}

public double CalculateTotalOrderValue()

{

double total = 0;

foreach (var product in Products)

{

total += product.Price; // Предполагаем, что количество каждого продукта в заказе равно 1

}

return total;

}

public string GetOrderSummary()

{

string productDetails = string.Join(", ", products.ConvertAll(p => p.GetProductSummary()));

return $"Заказ ID: {OrderID}, " +

$"Клиент: {Customer.Name}, " +

$"Товары: {productDetails}, " +

$"Дата: {OrderDate}, " +

$"Статус: {Status}, " +

$"Общая сумма: {CalculateTotalOrderValue()}";

}

public override string ToString()

{

return GetOrderSummary();

}

public object Clone()

{

var order = new Order(OrderID, (Customer)Customer.Clone());

var products = new List<Product>();

foreach (var product in Products)

{

order.AddProduct((Product)product.Clone());

}

order.OrderDate = OrderDate;

order.Status = Status;

return order;

}

public bool Equals(Order? other)

{

if (other == null)

return false;

return other.OrderID == OrderID;

}

public override bool Equals(object obj)

{

return Equals(obj as Order);

}

public override int GetHashCode()

{

return OrderID.GetHashCode();

}

public IEnumerator<Product> GetEnumerator()

{

return Products.GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable)Products).GetEnumerator();

}

}

**Шаг 6. Настройка главного класса**Program

1. В Program.cs создайте:
   * Несколько товаров (Product).
   * Клиента (Customer).
   * Заказ (Order).
2. Добавьте товары в заказ (order.AddProduct()).
3. Обновите статус заказа (order.UpdateStatus()).
4. Добавьте заказ клиенту (customer.AddOrder()).
5. Выведите информацию:
   * О клиенте (customer.GetCustomerInfo()).
   * О заказе (order.GetOrderSummary()).

// Главный класс программы

public class Program

{

public static void Main(string[] args)

{

// Создаем продукты

Product product1 = new Product("P001", "Ноутбук", 799.99, 10);

Product product2 = new Product("P002", "Смартфон", 499.99, 20);

// Создаем клиента

Customer customer = new Customer("C001", "Иван Иванов", "ivan@example.com", "Москва, ул. Пушкина, д. 1");

// Создаем заказ

Order order = new Order("O001", customer);

// Добавляем продукты в заказ

if (order.AddProduct(product1))

{

Console.WriteLine($"Продукт '{product1.Name}' добавлен в заказ.");

}

else

{

Console.WriteLine($"Не удалось добавить продукт '{product1.Name}' в заказ.");

}

if (order.AddProduct(product2))

{

Console.WriteLine($"Продукт '{product2.Name}' добавлен в заказ.");

}

else

{

Console.WriteLine($"Не удалось добавить продукт '{product2.Name}' в заказ.");

}

// Обновляем статус заказа

order.UpdateStatus(OrderStatus.Shipped);

// Добавляем заказ к клиенту

customer.AddOrder(order);

// Выводим информацию о клиенте

Console.WriteLine(customer.GetCustomerInfo());

// Проверяем, есть ли у клиента заказы

Console.WriteLine($"У клиента заказы: {customer.HasOrders()}");

// Выводим сводку заказа

Console.WriteLine(order.GetOrderSummary());

}

}

**Шаг 7. Тестирование программы**

1. Запустите программу (**F5**).
2. Проверьте:
   * Уменьшается ли количество товара при добавлении в заказ.
   * Корректно ли отображаются данные клиента и заказа.
   * Правильно ли меняется статус заказа.

**Контрольные вопросы**

1. Что такое инкапсуляция?
2. Что такое свойство?
3. Что такое полиморфизм?
4. Что такое наследование?
5. Как при вызове конструктора класса-наследника передать управление конструктору базового класса?
6. Как при вызове одного конструктора класса передать управление другому конструктору класса?