**Практическая работа №2**

**Оглавление**

[Практическая работа № 2. Интерфейсы. Метод расширение. Делегаты. 3](#_Toc163812652)

[Содержание пояснительной записки 24](#_Toc163812653)

[Используемое программное обеспечение 25](#_Toc163812654)

[Список литературы 26](#_Toc163812655)

Практическая работа № 2.  
Интерфейсы. Метод расширение. Делегаты.

**Цель практической работы**

Закрепление теоретических знаний по разработке объектно-ориентированных программ.

**Постановка задачи**

**Интерфейс**

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

**Определение интерфейса**

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

public interface IMovable

{

void Move();

}

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к статическим методам (не обязательно приватным):

interface IMovable

{

public const int minSpeed = 0;// минимальная скорость

private static int maxSpeed = 60;// максимальная скорость

// находим время, за которое надо пройти расстояние distance со скоростью speed

static double GetTime(double distance, double speed) => distance / speed;

static int MaxSpeed

{

get => maxSpeed;

set

{

if (value > 0) maxSpeed = value;

}

}

}

Console.WriteLine(IMovable.MaxSpeed); // 60

IMovable.MaxSpeed = 65;

Console.WriteLine(IMovable.MaxSpeed); // 65

double time = IMovable.GetTime(500, 10);

Console.WriteLine(time); // 50

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах.

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, реализуем выше определенный интерфейс IMovable:

interface IMovable

{

void Move();

}

// применение интерфейса в классе

class Person : IMovable

{

public void Move()

{

Console.WriteLine("Человек идет");

}

}

// применение интерфейса в структуре

struct Car : IMovable

{

public void Move()

{

Console.WriteLine("Машина едет");

}

}

IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя

IMovable car = new Car(); // Так можно

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

void DoAction(IMovable movable) => movable.Move();

interface IMovable

{

void Move();

}

class Person : IMovable

{

public void Move() => Console.WriteLine("Человек идет");

}

struct Car : IMovable

{

public void Move() => Console.WriteLine("Машина едет");

}

Person person = new Person();

Car car = new Car();

DoAction(person);

DoAction(car);

**IEnumerable, IEnumerator**

IEnumerable в C# представляет собой интерфейс, который определяет метод GetEnumerator(), возвращающий объект IEnumerator. Этот интерфейс позволяет перечислять элементы в коллекции, не раскрывая детали реализации самой коллекции.

Вот как работает IEnumerable в C#:

**1. Интерфейс IEnumerable**

* IEnumerable находится в пространстве имен System.Collections.
* Он определяет один метод GetEnumerator(), который возвращает объект IEnumerator.

**2. Интерфейс IEnumerator**

* IEnumerator также находится в пространстве имен System.Collections.
* Он определяет три основных члена:
  + Current: Получает текущий элемент в коллекции.
  + MoveNext(): Перемещает указатель на следующий элемент в коллекции.
  + Reset(): Сбрасывает указатель на начало коллекции.

**3. Реализация IEnumerable**

* Для использования IEnumerable в собственных коллекциях необходимо реализовать метод GetEnumerator().
* Метод GetEnumerator() должен возвращать объект, реализующий интерфейс IEnumerator.
* В методе GetEnumerator() создается экземпляр класса, реализующего IEnumerator, который будет использоваться для перечисления элементов коллекции.

Благодаря такой реализации мы можем 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 перечисления внутренних элементов коллекции.

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

}

**Важно знать:**

* IEnumerable позволяет работать с коллекциями в универсальной манере, не завися от типа конкретной коллекции.
* IEnumerable используется для реализации итераций по элементам коллекции без необходимости знать внутреннюю структуру коллекции.

Использование IEnumerable в C# помогает сделать ваш код более гибким и универсальным при работе с коллекциями и перечислениями.

**Пример.** Запишем класс для перечисления чисел Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 11 …

Данная последовательность бесконечная, как это можно реализовать в C#, ведь нельзя создать массив бесконечной длины?

В рамках примера начнем последовательность с 1.

public class FibonachiEnumerator : IEnumerator<double>

{

private double previous = 0, current = 1;

public double Current => current;

object IEnumerator.Current => current;

public void Dispose()

{

// Освобождение памяти при удалении объекта Fibonachi

}

// Переход к следующему элементу

public bool MoveNext()

{

var next = current + previous;

previous = current;

current = next;

return true;

}

// Сброс к начальному значению

public void Reset()

{

previous = 0;

current = 1;

}

}

Создадим класс, использующий данный перечислитель: Числа будут выводиться до бесконечности (пока не переполнится Double).

var fibonachi = new Fibonachi();

foreach (var f in fibonachi)

{

Console.Write($"{f} ");

Thread.Sleep(1000); // Задержка 1 секунда

}

public class Fibonachi : IEnumerable<double>

{

public IEnumerator GetEnumerator()

{

return new FibonachiEnumerator();

}

IEnumerator<double> IEnumerable<double>.GetEnumerator()

{

return (IEnumerator<double>)GetEnumerator();

}

}

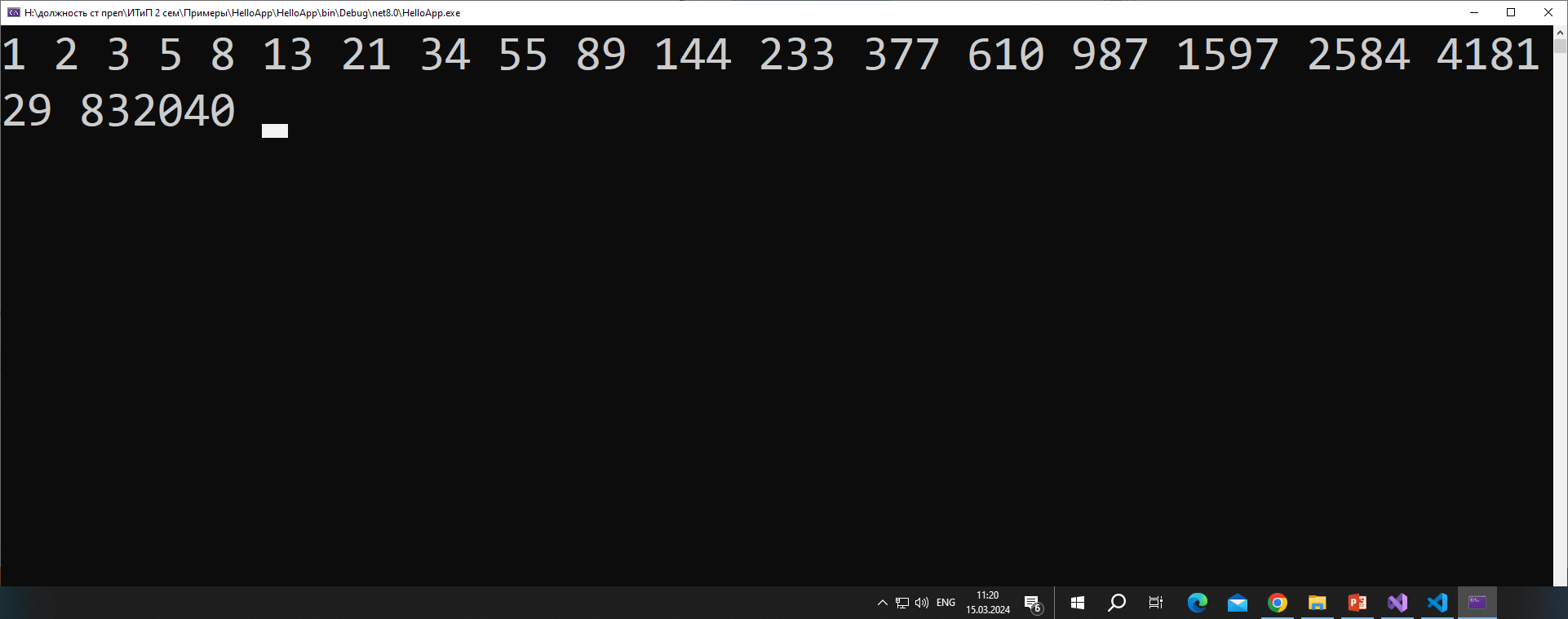


Рисунок 1 – Отображение чисел Фибоначчи

На основе данного класса можно создать коллекцию чисел Фибоначчи произвольной длины:

using System.Collections;

var fibonachi = new Fibonachi();

var numbers = fibonachi.Take(100).ToList();

С помощью метода Take запрашиваем 100 элементов, с помощью ToList добавляем их в список.

Тип **IEnumerable** не стоит путать с коллекциями, например, List или массивом. **IEnumerable** не имеет размера и не содержит объекты. Он предназначен для получения объектов по требованию.

var numbers = fibonachi.Take(100);

Вопрос какие объекты будут лежать в numbers?

Ответ: **никакие.** numbers имеет тип **IEnumerable<double>** это означает, что **numbers** это перечисление, которое по запросу будет возвращать объекты типа **double**. Получать объекты оно будет **находу** из **fibonachi**, никаких заранее созданных массивов для хранения объектов здесь нет!

Чтобы запросить объекты, нужно написать, например, **ToList**, или обратиться в цикле **foreach** и т.п.

С типом **IEnumerable** тесно связан оператор **yield return**. Данный оператор означает, что при последующем вызове метода мы зайдем на том месте, где вышли. Оператор **yield** может использоваться только в методах, возвращающих перечислитель, например, IEnumerable.

Рассмотрим пример с перечислением простых чисел:

public class Primes : IEnumerable<double>

{

public IEnumerator GetEnumerator()

{

yield return 2;

// При следующем вызове зайдем в метод здесь

int current = 2;  
 while (true)

{

current++;  
 if (!isPrime(current)) continue;  
 yield return current;

// При следующем вызове зайдем в метод здесь

}

}

private bool isPrime(int number)

{

int sqrt = (int)Math.Sqrt(number);

for (int i = 2; i <= sqrt; i++)

{

if (number % i == 0)

return false;

}

return true;

}

IEnumerator<double> IEnumerable<double>.GetEnumerator()

{

return (IEnumerator<double>)GetEnumerator();

}

}

**Пример.**

Пусть у нас есть класс Company, которая представляет компанию и которая хранит в массиве personnel штат сотрудников - объектов Person. Используем оператор **yield** для перебора этой коллекции.

class Person

{

public string Name { get; }

public Person(string name) => Name = name;

}

class Company

{

private Person[] personnel; // Сотрудники

public Company(Person[] personnel)

{

this.personnel = personnel;

}

public int Length => personnel.Length; // Кол-во сотрудников

public IEnumerator<Person> GetEnumerator()

{

for (int i = 0; i < personnel.Length; i++)

{// При итерировании компании возвращаем сотрудников

yield return personnel[i];

}

}

}

Воспользуемся разработанным классом:

var people = new Person[]

{

new Person("Tom"),

new Person("Bob"),

new Person("Sam")

};

var microsoft = new Company(people);

foreach (Person employee in microsoft)

{

// Итерируем компанию, в результате на каждой итерации блок yield return возвращает нам сотрудника employee

Console.WriteLine(employee.Name);

}

**Метод расширение**

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

public static class StringExtension

{

public static int CharCount(this string str,char c)

{

int counter = 0;

for (int i = 0; i < str.Length; i++)

{

if (str[i] == c)

counter++;

}

return counter;

}

}

string s = "Привет мир";

char c = 'и';

int i = s.CharCount(c);

Console.WriteLine(i);

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. В данном случае это класс **StringExtension**. Затем объявляем статический метод.

Суть нашего метода расширения - подсчет количества определенных символов в строке.

**Пример** методов расширение для поиска минимума и максимума в массиве:

public static class ArrayExtensions

{

/// <summary>

/// Метод расширение для поиска минимума

/// </summary>

/// <param name="arr">Массив чисел</param>

/// <returns>Минимум</returns>

public static double Min(this double[] arr)

{

var min = arr[0];

for (var i = 1; i < arr.Length; i++)

{

if (arr[i] < min)

min = arr[i];

}

return min;

}

/// <summary>

/// Метод расширение для поиска максимума

/// </summary>

/// <param name="arr">Массив чисел</param>

/// <returns>Максимум</returns>

public static double Max(this double[] arr)

{

var max = arr[0];

for (var i = 1; i < arr.Length; i++)

{

if (arr[i] > max)

max = arr[i];

}

return max;

}

}

var arr = new double[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Вызовем методы расширение

Console.WriteLine(arr.Min());

Console.WriteLine(arr.Max());

Метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра, то есть в нашем случае **this string str**. Так как наш метод будет относиться к типу string, то мы и используем данный тип.

**Делегаты (delegate)**

**Делегаты** представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Message mes; // 2. Создаем переменную делегата

mes = Hello; // 3. Присваиваем этой переменной адрес метода

mes(); // 4. Вызываем метод

void Hello() => Console.WriteLine("Hello METANIT.COM");

delegate void Message(); // 1. Объявляем делегат

**Пример. Делегат возвращающий значение.**

Operation operation = Add; // делегат указывает на метод Add

int result = operation(4, 5); // фактически Add(4, 5)

Console.WriteLine(result); // 9

operation = Multiply; // теперь делегат указывает на метод Multiply

result = operation(4, 5); // фактически Multiply(4, 5)

Console.WriteLine(result); // 20

int Add(int x, int y) => x + y;

int Multiply(int x, int y) => x \* y;

delegate int Operation(int x, int y);

**Пример 4.** Присвоение ссылки на метод.

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

Operation operation1 = Add;

Operation operation2 = new Operation(Add);

int Add(int x, int y) => x + y;

delegate int Operation(int x, int y);

**Добавление методов в делегат**

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип.

Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот список не один, а несколько методов.

Для добавления методов в делегат применяется операция +=:

Message message = Hello;

message += HowAreYou;

message(); // вызываются оба метода - Hello и HowAreYou

void Hello() => Console.WriteLine("Hello");

void HowAreYou() => Console.WriteLine("How are you?");

delegate void Message();

**Вызов делегата**

В примерах выше делегат вызывался как обычный метод. Другой способ вызова делегата представляет метод Invoke():

Message mes = Hello;

mes.Invoke(); // Hello

Operation op = Add;

int n = op.Invoke(3, 4);

Console.WriteLine(n); // 7

void Hello() => Console.WriteLine("Hello");

int Add(int x, int y) => x + y;

delegate int Operation(int x, int y);

delegate void Message();

Если делегат принимает параметры, то в метод **Invoke** передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, т.е. в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

Message? mes;

//mes(); // ! Ошибка: делегат равен null

Operation? op = Add;

op -= Add; // делегат op пуст

int n = op(3, 4); // !Ошибка: делегат равен null

int Add(int x, int y) => x + y;

delegate int Operation(int x, int y);

delegate void Message();

Поэтому при вызове делегата всегда лучше проверять, не равен ли он **null**. Либо можно использовать метод **Invoke** и оператор условного **null**:

Message? mes = null;

mes?.Invoke(); // ошибки нет, делегат просто не вызывается

Operation? op = Add;

op -= Add; // делегат op пуст

int? n = op?.Invoke(3, 4); // ошибки нет, делегат просто не вызывается, а n = null

int Add(int x, int y) => x + y;

delegate int Operation(int x, int y);

delegate void Message();

**События**

**События** — это особый вид многоадресного делегата, который может вызываться только из класса (или производных классов) или структуры, где они объявлены.

Если другие классы или структуры подписываются на событие, их методы обработчиков событий будут вызываться, когда класс будет вызывать событие.

События сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

**Определение и вызов событий**

События объявляются в классе с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие:

delegate void AccountHandler(string message);

event AccountHandler Notify;

В данном случае вначале определяется делегат **AccountHandler**, который принимает один параметр типа string. Затем с помощью ключевого слова **event** определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

class Account

{

public delegate void AccountHandler(string message);

public event AccountHandler? Notify; // 1.Определение события

public Account(int sum) => Sum = sum;

public int Sum { get; private set; }

public void Put(int sum)

{

Sum += sum;

Notify?.Invoke($"На счет поступило: {sum}");// 2.Вызов

}

public void Take(int sum)

{

if (Sum >= sum)

{

Sum -= sum;

Notify?.Invoke($"Со счета снято: {sum}"); // 2.Вызов

}

else

{

Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;

}

}

}

Теперь с помощью события Notify мы уведомляем систему о том, что были добавлены средства и о том, что средства сняты со счета или на счете недостаточно средств.

**Добавление обработчика события**

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция +=:

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

Account account = new Account(100);

account.Notify += DisplayMessage; // Добавляем обработчик для события Notify

account.Put(20); // добавляем на счет 20

Console.WriteLine($"Сумма на счете: {account.Sum}");

account.Take(70); // пытаемся снять со счета 70

Console.WriteLine($"Сумма на счете: {account.Sum}");

account.Take(180); // пытаемся снять со счета 180

Console.WriteLine($"Сумма на счете: {account.Sum}");

void DisplayMessage(string message) => Console.WriteLine(message);

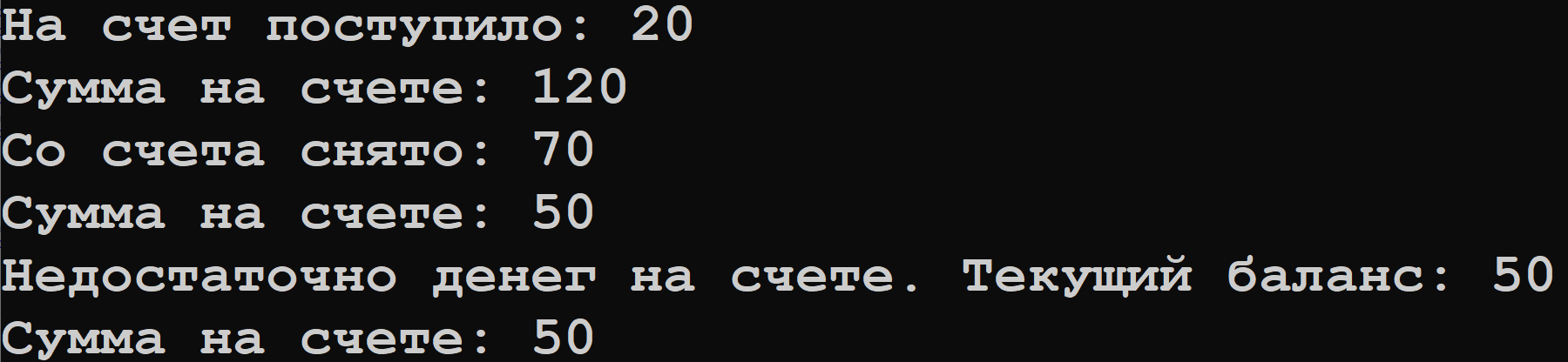


Рисунок 2 – Результат работы событий

**Передача данных события**

Нередко при возникновении события обработчику события требуется передать некоторую информацию о событии. Например, добавим и в нашу программу новый класс AccountEventArgs со следующим кодом:

class AccountEventArgs

{

// Сообщение

public string Message { get; }

// Сумма, на которую изменился счет

public int Sum { get; }

public AccountEventArgs(string message, int sum)

{

Message = message;

Sum = sum;

}

}

class Account

{

public delegate void AccountHandler(Account sender, AccountEventArgs e);

public event AccountHandler? Notify;

public int Sum { get; private set; }

public Account(int sum) => Sum = sum;

public void Put(int sum)

{

Sum += sum;

Notify?.Invoke(this, new AccountEventArgs($"На счет поступило {sum}", sum));

}

public void Take(int sum)

{

if (Sum >= sum)

{

Sum -= sum;

Notify?.Invoke(this, new AccountEventArgs($"Сумма {sum} снята со счета", sum));

}

else

{

Notify?.Invoke(this, new AccountEventArgs("Недостаточно денег на счете", sum));

}

}

}

По сравнению с предыдущей версией класса Account здесь изменилось только количество параметров у делегата и соответственно количество параметров при вызове события. Теперь делегат AccountHandler в качестве первого параметра принимает объект, который вызвал событие, то есть текущий объект Account. А в качестве второго параметра принимает объект AccountEventArgs, который хранит информацию о событии, получаемую через конструктор.

Теперь изменим основную программу:

Account acc = new Account(100);

acc.Notify += DisplayMessage;

acc.Put(20);

acc.Take(70);

acc.Take(150);

void DisplayMessage(Account sender, AccountEventArgs e)

{

Console.WriteLine($"Сумма транзакции: {e.Sum}");

Console.WriteLine(e.Message);

Console.WriteLine($"Текущая сумма на счете: {sender.Sum}");

}

По сравнению с предыдущим вариантом здесь мы только изменяем количество параметров и их использования в обработчике DisplayMessage. Благодаря первому параметру в методе можно получить информацию об отправителе события - счете, с которым производится операция. А через второй параметр можно получить информацию о состоянии операции.

**Задание на практическую работу**

Решаем практические задания с платформы **ulearn.me** тема «Наследование»:

<https://ulearn.me/course/basicprogramming/Postanovka_problemy_a969ed1f-fc3e-4dcc-8d05-6015069e3dcf>

(«Земля и Диггер», «Мешки и Золото», «Монстры»)

# 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»