# Министерство образования Республики Беларусь

УПРАВЛЕНИЕ ОБРАЗОВАНИЯ МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«Могилевский государственный политехнический колледж»

# Домашняя контрольная работа №1

По дисциплине: «**Конструирование программ и языки программирования**»

Группа ПО-455

## Выполнил О. Е. Листопадова

Шифр 21

# 2021

**27 Опишите логические операции.**

Следующие операторы выполняют логические операции с использованием логических операндов:

Унарный ! (логическое отрицание) оператор.

Бинарные & (логическое И), | (логическое ИЛИ), а также ^ (логическое исключающее ИЛИ) операторы. Эти операторы всегда обрабатывают оба операнда.

Бинарные && (условное логическое И) и || (условное логическое ИЛИ) операторы. Эти операторы вычисляют правый операнд, только если это необходимо.

Для операндов целочисленных типов операторы &, | и ^ выполняют побитовые логические операции. Дополнительные сведения см. в разделе Побитовые операторы и операторы сдвига.

**Оператор логического отрицания !**

Унарный префиксный оператор ! выполняет логическое отрицание операнда, возвращая true, если операнд имеет значение false, и false, если операнд имеет значение true.

bool passed = false;

Console.WriteLine(!passed); // output: True

Console.WriteLine(!true); // output: False

Начиная с версии C# 8.0, унарный постфиксный оператор !допускает значение NULL.

**Оператор логического И &**

Оператор & вычисляет логическое И для всех своих операндов. Результат операции x & y принимает значение true, если оба оператора x и y имеют значение true. В противном случае результат будет false.

Оператор & вычисляет оба операнда, даже если левый операнд имеет значение false. При этом операция должна вернуть значение false, независимо от значения правого операнда.

В следующем примере правый операнд оператора & является вызовом метода, который выполняется независимо от значения левого операнда:

bool SecondOperand()

{

Console.WriteLine("Second operand is evaluated.");

return true;

}

bool a = false & SecondOperand();

Console.WriteLine(a);

// Output:

// Second operand is evaluated.

// False

bool b = true & SecondOperand();

Console.WriteLine(b);

// Output:

// Second operand is evaluated.

// True

Условный оператор логического И && также вычисляет логическое И для своих операндов, но не вычисляет правый операнд, если левый операнд имеет значение false.

Для операндов целочисленных типов оператор & вычисляет побитовое логическое И своих операндов. Унарный оператор & является оператором AddressOf.

**Оператор логического исключения ИЛИ ^**

Оператор ^ вычисляет логическое исключение ИЛИ для всех своих операндов, возвращая true для x ^ y, если x имеет значение true и y имеет значение false или x имеет значение false и y имеет значение true. В противном случае результат будет false. То есть для операндов bool оператор ^ возвращает тот же результат, что и оператор неравенства !=.

Console.WriteLine(true ^ true); // output: False

Console.WriteLine(true ^ false); // output: True

Console.WriteLine(false ^ true); // output: True

Console.WriteLine(false ^ false); // output: False

Для операндов целочисленных типов оператор ^ вычисляет побитовое исключающее ИЛИ своих операндов.

**Оператор логического ИЛИ |**

Оператор | вычисляет логическое ИЛИ для всех своих операндов. Результат операции x | y принимает значение true, если хотя бы один из операторов x или y имеет значение true. В противном случае результат будет false.

Оператор | вычисляет оба операнда, даже если левый операнд имеет значение true. При этом операция должна вернуть значение true, независимо от значения правого операнда.

В следующем примере правый операнд оператора | является вызовом метода, который выполняется независимо от значения левого операнда:

bool SecondOperand()

{

Console.WriteLine("Second operand is evaluated.");

return true;

}

bool a = true | SecondOperand();

Console.WriteLine(a);

// Output:

// Second operand is evaluated.

// True

bool b = false | SecondOperand();

Console.WriteLine(b);

// Output:

// Second operand is evaluated.

// True

Условный оператор логического ИЛИ || также вычисляет логическое ИЛИ для своих операндов, но не вычисляет правый операнд, если левый операнд имеет значение true.

Для операндов целочисленных типов оператор | вычисляет побитовое логическое ИЛИ своих операндов.

**Условный оператор логического И &&**

Условный оператор логического И && (оператор короткого замыкания) вычисляет логическое И для своих операндов. Результат операции x && y принимает значение true, если оба оператора x и y имеют значение true. В противном случае результат будет false. Если x имеет значение false, y не вычисляется.

В следующем примере правый операнд оператора && является вызовом метода, который не выполняется, если левый операнд имеет значение false:

bool SecondOperand()

{

Console.WriteLine("Second operand is evaluated.");

return true;

}

bool a = false && SecondOperand();

Console.WriteLine(a);

// Output:

// False

bool b = true && SecondOperand();

Console.WriteLine(b);

// Output:

// Second operand is evaluated.

// True

Оператор логического И & также вычисляет логическое И для своих операндов, но он всегда вычисляет оба операнда.

**Условный оператор логического ИЛИ ||**

Условный оператор логического ИЛИ || (оператор короткого замыкания) вычисляет логическое ИЛИ для своих операндов. Результат операции x || y принимает значение true, если хотя бы один из операторов x или y имеет значение true. В противном случае результат будет false. Если x имеет значение true, y не вычисляется.

В следующем примере правый операнд оператора || является вызовом метода, который не выполняется, если левый операнд имеет значение true:

bool SecondOperand()

{

Console.WriteLine("Second operand is evaluated.");

return true;

}

bool a = true || SecondOperand();

Console.WriteLine(a);

// Output:

// True

bool b = false || SecondOperand();

Console.WriteLine(b);

// Output:

// Second operand is evaluated.

// True

Оператор логического ИЛИ | также вычисляет логическое ИЛИ для своих операндов, но всегда вычисляет оба операнда.

**49 Опишите механизм событий.**

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

Например, возьмем следующий класс, который описывает банковский счет:

class Account

{

public Account(int sum)

{

Sum = sum;

}

// сумма на счете

public int Sum { get; private set;}

// добавление средств на счет

public void Put(int sum)

{

Sum += sum;

}

// списание средств со счета

public void Take(int sum)

{

if (Sum >= sum)

{

Sum -= sum;

}

}

}

В конструкторе устанавливаем начальную сумму, которая хранится в свойстве Sum. С помощью метода Put мы можем добавить средства на счет, а с помощью метода Take, наоборот, снять деньги со счета. Попробуем использовать класс в программе - создать счет, положить и снять с него деньги:

static void Main(string[] args)

{

Account acc = new Account(100);

acc.Put(20); // добавляем на счет 20

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

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

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

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

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

Console.Read();

}

Консольный вывод:

Сумма на счете: 120

Сумма на счете: 50

Сумма на счете: 50

Все операции работают как и положено. Но что если мы хотим уведомлять пользователя о результатах его операций. Мы могли бы, например, для этого изменить метод Put следующим образом:

public void Put(int sum)

{

Sum += sum;

Console.WriteLine($"На счет поступило: {sum}");

}

Казалось, теперь мы будем извещены об операции, увидев соответствующее сообщение на консоли. Но тут есть ряд замечаний. На момент определения класса мы можем точно не знать, какое действие мы хотим произвести в методе Put в ответ на добавление денег. Это может вывод на консоль, а может быть мы захотим уведомить пользователя по email или sms. Более того мы можем создать отдельную библиотеку классов, которая будет содержать этот класс, и добавлять ее в другие проекты. И уже из этих проектов решать, какое действие должно выполняться. Возможно, мы захотим использовать класс Account в графическом приложении и выводить при добавлении на счет в графическом сообщении, а не консоль. Или нашу библиотеку классов будет использовать другой разработчик, у которого свое мнение, что именно делать при добавлении на счет. И все эти вопросы мы можем решить, используя 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.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

Notify("Произошло действие");

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

if(Notify !=null) Notify("Произошло действие");

Или так:

Notify?.Invoke("Произошло действие");

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода Invoke(), передав в него необходимые значения для параметров.

Объединим все вместе и создадим и вызовем событие:

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 события

}

public void Take(int sum)

{

if (Sum >= sum)

{

Sum -= sum;

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

}

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 снты 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 +=:

Notify += обработчик события;

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

class Program

{

static void Main(string[] args)

{

Account acc = new Account(100);

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

acc.Put(20); // добавляем на счет 20

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

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

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

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

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

Console.Read();

}

private static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

}

В данном случае в качестве обработчика используется метод DisplayMessage, который соответствует по списку параметров и возвращаемому типу делегату AccountHandler. В итоге при вызове события Notify?.Invoke() будет вызываться метод DisplayMessage, которому для параметра message будет передаваться строка, которая передается в Notify?.Invoke(). В DisplayMessage просто выводим полученное от события сообщение, но можно было бы определить любую логику.

Если бы в данном случае обработчик не был бы установлен, то при вызове события Notify?.Invoke() ничего не происходило, так как событие Notify было бы равно null.

Консольный вывод программы:

На счет поступило: 20

Сумма на счете: 120

Со счета снято: 70

Сумма на счете: 50

Недостаточно денег на счете. Текущий баланс: 50

Сумма на счете: 50

Теперь мы можем выделить класс Account в отдельную библиотеку классов и добавлять в любой проект.

**Добавление и удаление обработчиков**

Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция -=. Например:

class Program

{

static void Main(string[] args)

{

Account acc = new Account(100);

acc.Notify += DisplayMessage; // добавляем обработчик DisplayMessage

acc.Notify += DisplayRedMessage; // добавляем обработчик DisplayMessage

acc.Put(20); // добавляем на счет 20

acc.Notify -= DisplayRedMessage; // удаляем обработчик DisplayRedMessage

acc.Put(20); // добавляем на счет 20

Console.Read();

}

private static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

private static void DisplayRedMessage(String message)

{

// Устанавливаем красный цвет символов

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine(message);

// Сбрасываем настройки цвета

Console.ResetColor();

}

}

Консольный вывод:

На счет поступило: 20

На счет поступило: 20

На счет поступило: 20

**Управление обработчиками**

С помощью специальных акссесоров add/remove мы можем управлять добавлением и удалением обработчиков. Как правило, подобная функциональность редко требуется, но, тем не менее мы ее можем использовать. Например:

class Account

{

public delegate void AccountHandler(string message);

private event AccountHandler \_notify;

public event AccountHandler Notify

{

add

{

\_notify += value;

Console.WriteLine($"{value.Method.Name} добавлен");

}

remove

{

\_notify -= value;

Console.WriteLine($"{value.Method.Name} удален");

}

}

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}");

}

public void Take(int sum)

{

if (Sum >= sum)

{

Sum -= sum;

\_notify?.Invoke($"Со счета снято: {sum}");

}

else

{

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

}

}

}

Теперь опредление события разбивается на две части. Вначале просто определяется переменная, через которую мы можем вызывать связанные обработчики:

private event AccountHandler \_notify;

Во второй части определяем акссесоры add и remove. Аксессор add вызывается при добавлении обработчика, то есть при операции +=. Добавляемый обработчик доступен через ключевое слово value. Здесь мы можем получить информацию об обработчике (например, имя метода через value.Method.Name) и определить некоторую логику. В данном случае для простоты просто выводится сообщение на консоль:

add

{

\_notify += value;

Console.WriteLine($"{value.Method.Name} добавлен");

}

Блок remove вызывается при удалении обработчика. Аналогично здесь можно задать некоторую дополнительную логику:

remove

{

\_notify -= value;

Console.WriteLine($"{value.Method.Name} удален");

}

Внутри класса событие вызывается также через переменную \_notify. Но для добавления и удаления обработчиков в программе используется как раз Notify:

class Program

{

static void Main(string[] args)

{

Account acc = new Account(100);

acc.Notify += DisplayMessage; // добавляем обработчик DisplayMessage

acc.Put(20); // добавляем на счет 20

acc.Notify -= DisplayMessage; // удаляем обработчик DisplayRedMessage

acc.Put(20); // добавляем на счет 20

Console.Read();

}

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

}

Консольный вывод программы:

DisplayMessage добавлен

На счет поступило: 20

DisplayMessage удален

**62 Напишите программу, которая определяет максимальное число из введенной с клавиатуры последовательности положительных чисел (длина последовательности неограниченна). Ниже приведен ре­комендуемый вид экрана во время выполнения программы (данные, введенные пользователем, выделены полужирным шрифтом). Определение максимального числа последовательности положительных чисел.**

Пример: Вводите после стрелки числа. Для завершения ввода введите ноль,

- > 56

- > 75

- >43

- >0

Максимальное число: 75

**Блок-схема алгоритма решения задачи.**

**Текст программы.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp62

{

class Program

{

static void Main(string[] args)

{

int n;// число введенное с клавиатуры

//

int m = 0; // максимальное число

Console.WriteLine("Вычисление максимального числа последовательности");

Console.WriteLine("Для окончания введите 0");

do

{

Console.Write("->");

n = Convert.ToInt32(Console.ReadLine());

if (n > m)

{

m = n;

}

}

while (n > 0);

Console.WriteLine("Максимальное число: " + m);

Console.WriteLine("Нажмите ВВОД ");

Console.ReadLine();

}

}

}

**84 Решите задачу. Дан двумерный массив. Заполните его по строкам с клавиатуры. Отсортируйте каждый столбец по возрастанию.**

**Блок-схема алгоритма решения задачи.**

**Текст программы.**

**Список использованных источников**

1. Албахари, Д. C# 5.0. Справочник. Полное описание языка / Д.Албахари. – М.: Вильямс, 2014.
2. Лабор, В.В. Си Шарп: Создание приложений для Windows / В.В.Лабор. - Мн.: Харвест, 2003.
3. Павловская, Т.А С#. Программирование на языке высокого уровня / Т.А.Павловская. - СПб: Питер, 2014.
4. Фролов, А.В. Визуальное проектирование приложений С# / А.В.Фролов. - М: КУДИЦ - ОБРАЗ, 2003.
5. Фленов, М. Библия С# / М.Фленов. - СПб.: Питер, 2011.