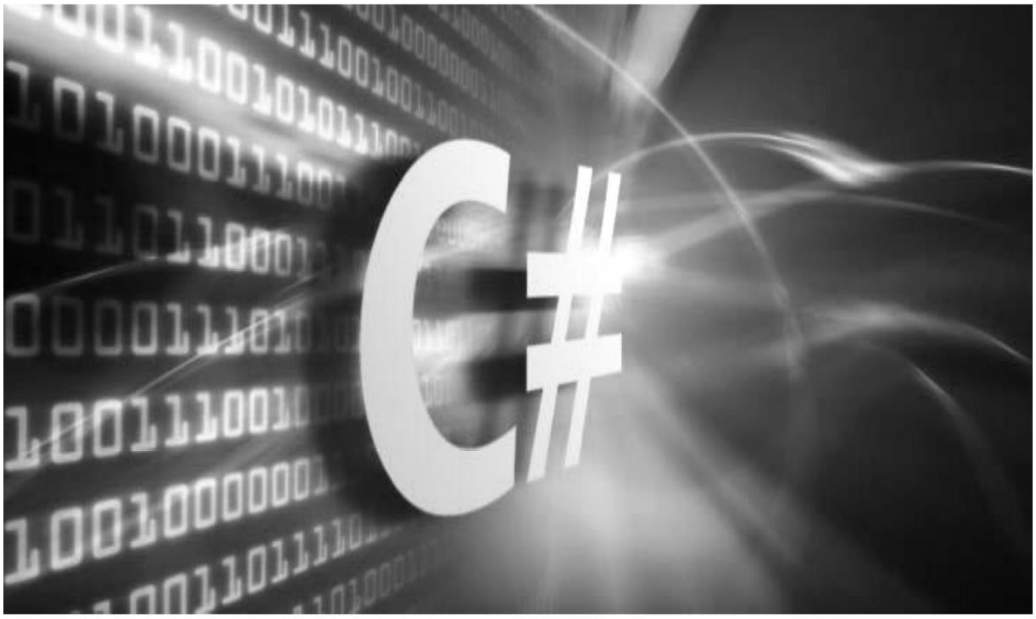
УНИВЕРСИТЕТ ИТМО

Н.А. Осипов

Технологии программирования

Учебное пособие



Санкт-Петербург

Оглавление

Введение **5**

Лабораторная работа 1. Создание программы с помощью среды разработки Visual Studio.NET **5**

Упражнение 1. Создание простой программы в текстовом редакторе 5

Упражнение 2. Создание программы с помощью среды разработки Visual Studio .NET 7

Упражнение 3. Использование отладчика Visual Studio .NET 8

Упражнение 4. Добавление в C#-программу обработчика исключительных ситуаций 9

Упражнение 5. Расчет площади треугольника 12

Лабораторная работа 2. Создание и использование размерных типов данных **12**

Упражнение 1. Создание перечисления 12

Упражнение 2. Создание и использование структуры 13

Упражнение 3. Реализация структуры Distance 14

Лабораторная работа 3. Использование выражений **15**

Упражнение 1. Реализация операторов выбора 15

Упражнение 2. Реализация циклов при работе с данными размерных типов 17

Лабораторная работа 4. Создание и использование методов **20**

Упражнение 1. Использование параметров в методах, возвращающих значения 20

Упражнение 2. Использование в методах параметров, передаваемых по ссылке 21

Упражнение 3. Использование возвращаемых параметров в методах 22

Упражнение 4. Расчет площади треугольника с помощью метода 24

Упражнение 5. Вычисление корней квадратного уравнения 25

Лабораторная работа 5. Создание и использование массивов **25**

Упражнение 1. Работа с массивом размерного типа данных 25

Упражнение 2. Перемножение матриц 26

Упражнение 3. Обработка данных массива 31

Лабораторная работа 6. Создание и использование классов **31**

Упражнение 1. Разработка класса Book 31

Упражнение 2. Использование конструкторов 33
Упражнение 3. Реализация класса Triangle 34

Лабораторная работа 7. Создание иерархии классов 35
Упражнение 1. Реализация наследования классов 35
Упражнение 2. Использование конструкторов 37
Упражнение 3. Переопределение методов 41
Упражнение 4. Применение абстрактного класса и абстрактных методов 42
Упражнение 5. Реализации модели включения 43
Упражнение 6. Реализация отношения ассоциации между классами 45
Упражнение 7. Реализация прогрессии 48

Лабораторная работа 8. Использование интерфейсов при реализации иерархии классов 49
Упражнение 1. Создание и реализация интерфейса 49
Упражнение 2. Использование стандартных интерфейсов 50
Упражнение 3. Реализация прогрессии с помощью интерфейса 51

Лабораторная работа 9. Применение делегатов и событий ... 52
Упражнение 1. Использование делегата при вызове метода 52
Упражнение 2. Работа с событиями 53
Упражнение 3. Реализация события 55

Задание на самостоятельную работу. Иерархия классов учебного центра 55

Приложение 56
Таблица 1. Параметры форматирования C# 56
Таблица 2. Флаги компиляции компилятора командной строки C# 57

Список литературы 58

Лабораторная работа 1. Создание программы с помощью среды разработки Visual Studio.NET

Цель работы

Изучение структуры программы на языке C# и приобретение навыков ее компиляции и отладки.

Упражнение 1. Создание простой программы в текстовом редакторе

В этом упражнении вы напишите программу на языке C#, используя обычный текстовый редактор, например, Блокнот. В окне командной строки программа будет спрашивать, как вас зовут и затем здороваться с вами по имени.

➢ **Создайте текст программы**

- Откройте текстовый редактор – Блокнот.
- В редакторе введите директиву, которая разрешает использовать имена стандартных классов из пространства имен **System** непосредственно без указания имени пространства, в котором они были определены:

```
using System;
```

- Язык C# является объектно-ориентированным языком и написанные на нем программы представляют собой совокупность взаимодействующих между собой классов. Добавьте класс с именем **Program:**

```
class Program
{

}
```

- В классе определите метод **Main,** укажите ключевое слово **static**, которое означает что это метод можно вызывать, не создавая объект типа **Program** и тип возвращаемого значения **void** (который означает, что метод не возвращает значение):

```
class Program
   {
        static void Main()
        {

        }

   }
```

- В методе **Main** вставьте следующую строку кода:
  ```
  string myName;
  ```
- Напишите код, запрашивающий имя пользователя:
  ```
  Console.WriteLine("Please enter your name");
  ```
- Напишите код, считывающий введенное пользователем имя и присваивающий полученное значение строковой переменной *myName:*
  ```
  myName = Console.ReadLine( );
  ```
- Добавьте код, который будет выводить на экран строку “Hello *myName*”, где *myName* – имя, введенное пользователем.
  ```
  Console.WriteLine("Hello, {0}", myName);
  ```
- Итоговый текст метода **Main** должен выглядеть следующим образом:
  ```
  static void Main()
  {
  string myName;
  Console.WriteLine("Please enter your name");
  myName = Console.ReadLine( );
  Console.WriteLine("Hello, {0}", myName);
  }
  ```
- Сохраните текст программы в папке проектов C:\Users\пользователь\Documents\Visual Studio\Projects под именем MyProgram.cs (чтобы сохранить именно с требуемым расширением следует ввести при сохранении файла в текстовом редакторе имя и расширение в кавычках).

➢ **Откомпилируйте и запустите Ваше приложение из командной строки**

- Запустите окно командной строки **Visual Studio .NET Command Prompt**. **(Start→All Programs→Visual Studio .NET→Visual Studio .NET Tools→Visual Studio .NET Command Prompt)**
- Изучите аргументы компилятора, для этого введите команду
  ```
  csc -?
  ```

и нажмите клавишу <**Enter**>. Если на компьютере развернута среда разработки .NET появится список аргументов командной строки, которые может принимать работающий в режиме командной строки компилятор C#.

- С помощью команды **cd** перейдите в каталог C:\Users\пользователь\Documents\Visual Studio \Projects.
- Откомпилируйте Вашу программу, используя следующую команду:
  ```
  csc /out:MyHelloProgram.exe MyProgram.cs
  ```

- Если в программе были допущены ошибки, то компилятор отобразит их на экране. При наличии ошибок откройте снова программу в текстовом редакторе и исправьте ошибки.
- Если ошибок нет запустите программу, набрав в командной строке ее название:
```
MyHelloProgram
```
- Введите свое имя и нажмите клавишу <**Enter**>.
- Закройте окно командной строки.

Упражнение 2. Создание программы с помощью среды разработки Visual Studio .NET

В этом упражнении вы напишите программу первого упражнения, используя среду разработки Visual Studio.NET.

➢ **Создайте новое консольное приложение C#**

- Запустите **Microsoft Visual Studio.NET. (Start→All Programs→Visual Studio .NET→ Microsoft Visual Studio.NET).**
- Выберите пункт меню **File→New→Project**
- На панели **Project Types** выберите **Visual C# Projects.**
- На панели **Templates** выберите **Console Application**.
- В текстовое поле ***Name*** введите имя проекта Greetings.
- В поле **Location** укажите каталог для проекта *install folder*\Labs\Lab01 и нажмите **OK.**
- Измените имя класса на **Greeter**.
- Сохраните проект, выбрав пункт меню **File→Save All.**

➢ **Напишите код, запрашивающий имя пользователя и приветствующий его по имени**

- В методе **Main** вставьте следующую строку кода:
```
string myName;
```
- Напишите код, запрашивающий имя пользователя:
```
Console.WriteLine("Please enter your name");
```
- Напишите код, считывающий введенное пользователем имя и присваивающий полученное значение строковой переменной *myName:*
```
myName = Console.ReadLine( );
```
- Добавьте код, который будет выводить на экран строку “Hello *myName*”, где *myName* – имя, введенное пользователем.
```
Console.WriteLine("Hello, {0}", myName);
```
- Итоговый текст метода **Main** должен выглядеть следующим образом:
```
static void Main(string[ ] args)
{
string myName;
Console.WriteLine("Please enter your name");
```

```
myName = Console.ReadLine( );
Console.WriteLine("Hello, {0}", myName);
}
```

- Сохраните проект.

➢ **Откомпилируйте и запустите программу**

- Выберите пункт меню **Build→Build Solution (**или **Ctrl+Shift+B).**
- При необходимости исправьте ошибки и откомпилируйте программу заново.
- Выберите пункт меню **Debug→Start Without Debugging (**или **Ctrl+F5).**
- В появившемся окне введите свое имя и нажмите **ENTER.**
- Закройте приложение.

Упражнение 3. Использование отладчика Visual Studio .NET

В этом задании вы научитесь работать с интегрированным отладчиком Visual Studio .NET: проходить программу по шагам и просматривать значения переменных.

➢ **Поставьте точки остановки и запустите пошаговое выполнение**

- Запустите **Visual Studio .NET,** если она не запущена.
- Выберите пункт меню **File→Open→Project.**
- Откройте проект Greetings.sln из папки *install folder*\Labs\Lab01\Greetings.
- В редакторе кода класса **Greeter** щелкните по крайнему левому полю на уровне строки кода, где впервые встречается команда **Console.WriteLine.**
- Выберите пункт меню **Debug→Start (**или нажмите **F5).**

Программа запустится на выполнение, появится консольное окно и затем программа прервется в месте точки остановки.

➢ **Просмотрите значение переменной**

- Выберите пункт меню **Debug →Windows→Watch→Watch1.** Обратите внимание, что эти окна доступны только в процессе отладки.
- В окне **Watch** в список выражений для мониторинга добавьте переменную *myName*.
- В окне **Watch** появится переменная *myName* с текущим значением **null.**

➢ **Используйте команды пошагового выполнения**

- Для выполнения первой команды **Console.WriteLine** выберите пункт меню **Debug→Step Over (**или нажмите **F10).**

- Для выполнения следующей строчки кода, содержащей команду **Console.ReadLine,** снова нажмите **F10.**
- Вернитесь в консольное окно, введите свое имя и нажмите **ENTER.**

Вернитесь в Visual Studio. Текущее значение переменной *myName* в окне **Watch** будет содержать ваше имя.

- Для выполнения следующей строчки кода, содержащей команду **Console. WriteLine,** снова нажмите **F10.**
- Разверните консольное окно. Там появилось приветствие.
- Вернитесь в Visual Studio. Для завершения выполнения программы выберите пункт меню **Debug→Continue (**или нажмите **F5).**

Упражнение 4. Добавление в C#-программу обработчика исключительных ситуаций

В этом упражнении вы напишите программу, в которой будет использоваться обработчик исключительных ситуаций, который будет отлавливать ошибки времени выполнения. Программа будет запрашивать у пользователя два целых числа, делить первое число на второе и выводить полученный результат.

➢ **Создайте новый проект**

- Запустите **Visual Studio .NET**.
- Выберите пункт меню **File→New→Project**
- На панели **Project Types** выберите **Visual C# Projects.**
- На панели **Templates** выберите **Console Application.**
- В текстовое поле ***Name*** введите имя проекта **Divider.**
- В поле **Location** укажите каталог для проекта *install folder*\Labs\Lab01 и нажмите **OK.**
- Измените имя класса на **DivideIt**.
- Сохраните проект, выбрав пункт меню **File→Save All.**

➢ **Напишите код, запрашивающий у пользователя два целых числа.**

- В методе **Main** напишите код, запрашивающий у пользователя первое целое число:

```
Console.WriteLine("Please enter the first integer");
```

- Напишите код, считывающий введенное пользователем число и присваивающий полученное значение переменной *temp* типа **string**:

```
string temp = Console.ReadLine( );
```

- Добавьте код, который переведет значение переменной *temp* из типа данных **string** в **int** и сохранит полученный результат в переменной *i*:

```
int i = Int32.Parse(temp);
```

- Аналогичным образом создайте следующий код:
  - Запросите у пользователя второе целое число.

- Считайте введенное пользователем число и присвойте полученное значение переменной *temp.*
- Переведите значение переменной *temp* в тип данных **int** и сохраните полученный результат в переменной *j*.

Итоговый текст программы должен выглядеть следующим образом:

```
Console.WriteLine("Please enter the first integer");
string temp = Console.ReadLine( );
int i = Int32.Parse(temp);
Console.WriteLine("Please enter the second integer");
temp = Console.ReadLine( );
int j = Int32.Parse(temp);
```

- Сохраните проект.

➢ **Разделите первое число на второе и выведите результат на экран**

- Напишите код, создающий новую переменную *k* типа **int**, в которую будет заноситься результат деления числа *i* на *j,* и поместите его после кода, созданного в предыдущем пункте.

```
int k = i / j;
```

- Добавьте код, выводящий значение *k* на экран.
- Сохраните проект.

➢ **Протестируйте программу**

- Выберите пункт меню **Debug→Start Without Debugging (**или **Ctrl+F5).**
- Введите первое число **10** и нажмите **ENTER.**
- Введите второе число **5** и нажмите **ENTER.**
- Проверьте, что выводимое значение *k* будет равным 2.

Обратите внимание на типы данных. Подумайте, почему при делении, например, 10 на 3 получается 3.

- Снова запустите программу на выполнение. Введите первое число **10** и нажмите **ENTER.** Введите второе число **0** и нажмите **ENTER.**
- В программе возникнет исключительная ситуация (деление на ноль).
- Выберите **Закрыть программу.**
- Закройте окно командной строки.

➢ **Добавьте в программу обработчик исключительных ситуаций**

- Поместите код метода **Main** внутрь блока **try** следующим образом:

```
try
{
Console.WriteLine (...);
...
int k = i / j;
Console.WriteLine(...);
}
```

- В методе **Main** после блока **try** добавьте блок **catch**, внутри которого должно выводиться краткое сообщение об ошибке:

```
catch(Exception e)
{
Console.WriteLine("An exception was thrown: {0}",
e.Message);
}
...
```

- Сохраните проект.
- Итоговый текст метода **Main** должен выглядеть следующим образом:

```
public static void Main(string[ ] args)
{
   try {
         Console.WriteLine ("Please enter the first integer");
         string temp = Console.ReadLine( );
         int i = Int32.Parse(temp);

         Console.WriteLine ("Please enter the second integer");
         temp = Console.ReadLine( );
         int j = Int32.Parse(temp);

         int k = i / j;
         Console.WriteLine("The result of dividing {0} by {1} is
    {2}", i, j, k);
       }
       catch(Exception e) {
      Console.WriteLine("An exception was thrown: {0}",
e.Message);
       }
}
```

➢ **Протестируйте код обработчика исключительных ситуаций**

- Снова запустите программу на выполнение, нажав **Ctrl+F5.**
- Введите первое число **10** и нажмите **ENTER.**
- Введите второе число **0** и нажмите **ENTER.**

В программе вновь возникнет исключительная ситуация (деление на ноль), но на этот раз ошибка перехватывается и на экран выводится сообщение.

- Повторите запуск программы и введите вместо числа букву. Изучите сообщение об исключительной ситуации.

➢ **Добавьте в программу обработчик исключительной ситуации при вводе данных неверного формата**

- Поместите код нового обработчика ошибки **перед** универсальным обработчиком:

```
catch (FormatException e) {
 Console.WriteLine("An format exception was thrown:
{0}", e.Message);
}
catch(Exception e) {
```

```
    Console.WriteLine("An exception was thrown: {0}",
e.Message);
    }
```

- Повторите запуск программы и введите вместо числа букву. В программе вновь возникнет исключительная ситуация (входная строка имеет неверный формат), но на этот раз ошибка перехватывается и на экран выводится соответствующее сообщение.

Упражнение 5. Расчет площади треугольника

Требуется создать программу, которая подсчитывает площадь равностороннего треугольника, периметр которого известен.

Реализуйте диалог с пользователем:

- с клавиатуры вводится значение периметра,
- после расчетов на экран выводится информация в виде небольшой таблицы:

Сторона	Площадь
Значение	*Результат*

- Результаты расчетов представьте в формате числа с двумя знаками после запятой.
  - ✓ *Рекомендация*. Можно воспользоваться параметрами форматирования C# (см. таблица 1 в приложении)

Для расчета площади используйте формулу Герона:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где p – полупериметр.

Метод вычисления квадратного корня определен в классе **Math** и может быть применен, например, следующим образом:

```
double S = Math.Sqrt(...);
```

Лабораторная работа 2. Создание и использование размерных типов данных

Цель работы

Изучение размерных типов, данных и приобретение навыков работы со структурными типами.

Упражнение 1. Создание перечисления

В этом упражнении вы создадите перечисление для представления различных типов банковских счетов. Затем вы используете данное перечисление для создания двух переменных, которым вы присвоите значения **Checking** и **Deposit**. Далее вы выведете на экран значения этих переменных, используя функцию **System.Console.WriteLine**.

➢ Создайте перечисление

- Создайте проект BankAccount.sln в папке
  install folder\Labs\Lab02\BankAccount.

- Переименуйте файл Program.cs на файл Enum.cs, согласитесь с предложением изменить ссылки на новое имя.
- Перед описанием класса добавьте перечисление **AccountType**:
```
public enum AccountType { Checking, Deposit }
```
Данное перечисление содержит значения **Checking** и **Deposit**.
- В методе **Main()** объявите две переменные типа **AccountType**:
```
AccountType goldAccount;
AccountType platinumAccount;
```
- Присвойте первой переменной значение Checking, а второй – Deposit:
```
goldAccount = AccountType.Checking;
platinumAccount = AccountType.Deposit;
```
- Выведите на консоль значения обеих переменных, два раза используя метод **Console.WriteLine**:
```
Console.WriteLine("The Customer Account Type is {0}",
goldAccount);
Console.WriteLine("The Customer Account Type is {0}",
platinumAccount);
```
- Откомпилируйте и запустите программу.

Упражнение 2. Создание и использование структуры

В этом упражнении вы создадите структуру, которую можно использовать для представления банковских счетов. Для хранения номеров счетов (тип данных **long)**, балансов счетов (тип данных **decimal**) и типов счетов (перечисление, созданное в упражнении 2) вы будете использовать переменные. Затем создадите переменную типа структуры, заполните ее данными и выведете результаты на консоль.

➢ **Создайте структуру**

- Создайте проект StructType.sln в папку *install folder*\Labs\Lab02\StructType.
- Переименуйте файл Program.cs на файл Struct.cs, согласитесь с предложением изменить ссылки на новое имя.
- Откройте файл Struct.cs и перед описанием класса добавьте перечисление **AccountType**:
```
public enum AccountType { Checking, Deposit }
```
Данное перечисление содержит типы **Checking** и **Deposit.**
- После перечисления добавьте **public** структуру **BankAccount**, содержащую следующие поля:

Модификатор доступа	**Тип**	**Переменная**
public	long	*accNo*
public	decimal	*accBal*
public	AccountType	*accType*

```
public struct BankAccount
{
    public long accNo;
    public decimal accBal;
```

```
        public AccountType accType;
    }
```

- В методе **Main() о**бъявите переменную типа **BankAccount**:

```
BankAccount goldAccount;
```

- Присвойте значения полям *accNo, accBal* и *accType* переменной *goldAccount.*

```
goldAccount.accType = AccountType.Checking;
goldAccount.accBal = (decimal)3200.00;
goldAccount.accNo = 123;
```

- Выведите на консоль значения каждого из элементов переменной структуры, используя инструкцию **Console.WriteLine**.

```
Console.WriteLine("*** Account Summary ***");
Console.WriteLine("Acct Number {0}",
goldAccount.accNo);
Console.WriteLine("Acct Type {0}",
goldAccount.accType);
Console.WriteLine("Acct Balance
${0}",goldAccount.accBal);
```

- Откомпилируйте и запустите программу.

➢ **Добавьте возможность ввода/вывода**

- Откройте (если он не открыт) проект StructType.sln из папки
  install folder\Labs\Lab02\ StructType.
- В файле Struct.cs замените следующую строку:

```
goldAccount.accNo = 123;
```

на инструкцию **Console.Write** для запроса номера банковского счета у пользователя:

```
Console.Write("Enter account number: ");
```

- Считайте номер счета, используя инструкцию **Console.ReadLine.** Присвойте полученное значение переменной *goldAccount.accNo.*

```
goldAccount.accNo = long.Parse(Console.ReadLine());
```

*Замечание***:** Перед тем как присвоить считанное значение переменной *goldAccount.accNo*, необходимо преобразовать его из типа **string** в тип **long**, используя метод **Long.Parse.**

- Откомпилируйте и запустите программу. При запросе введите номер счета.

Упражнение 3. Реализация структуры Distance

Требуется создать структуру **Distance,** определяющую длину в английской системе мер.

В английской системе мер основными единицами измерения длины служат фут и дюйм, причем один фут равен 12 дюймам. Расстояние, равное, например, 15 футам и 8 дюймам, на экран выведете как 15 '- 8". Дефис в данной записи будет служить для разделения значений футов и дюймов.

Для тестирования структуры **Distance** в методе **Main** класса **Program**:

- определите три переменные типа **Distance** и две из них инициализируйте с помощью значений, вводимых с клавиатуры;
- присвойте третьей переменной значение суммы первых двух переменных;
- пересчитайте сумму с учетом того, что один фут равен 12 дюймам (для футов примените операцию приведения типа, например, `(int)(Z.inch / 12);`; для дюймов – операцию получения остатка от деления, например, `Z.inch % 12`);
- выведите результат на экран согласно требуемого формата.

Лабораторная работа 3. Использование выражений

Цель работы

Изучение и приобретение навыков использования управляющих конструкций для организации вычислений.

Упражнение 1. Реализация операторов выбора

Задание 1. Применение конструкции if-else-if

В этом задании вы составите программу, которая выдает одно из сообщений «Да», «Нет», «На границе» в зависимости от того, лежит ли точка внутри заштрихованной области (см. рис. 3.1.1), вне заштрихованной области или на ее границе.

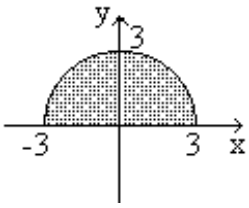


Рис. 3.1.1 Область определения функции

➢ **Создайте проект**

- Создайте проект Shapeifelse.sln в папке *install folder*\Labs\Lab03\.

➢ **Реализуйте ввод координат точки**

- В методе **Main** с помощью методов **Write** и **ReadLine** запросите у пользователя значения координат проверяемой точки:

```
Console.Write("x=");
float x = float.Parse(Console.ReadLine());
Console.Write("y=");
float y = float.Parse(Console.ReadLine());
```

- Проверьте попадает ли точка в область, используя конструкцию **if-else-if**:

```
if (x * x + y * y < 9 && y > 0)
    Console.WriteLine("внутри");
```

```
        else if (x * x + y * y > 9 || y < 0)
              Console.WriteLine("вне");
               else Console.WriteLine("на границе");
```

- Постройте и запустите приложение. Протестируйте работу программы.

Задание 2. Применение оператора switch

Вы создадите программу моделирующую работу калькулятора. Пользователь должен ввести первый операнд, затем требуемую операцию и второй операнд. В зависимости от знака операции будет проведен расчет результата.

➢ **Создайте проект**

- Создайте проект Calc_switch.sln в папке *install folder*\Labs\Lab03\.

➢ **Реализуйте ввод операндов и символа операции**

- В методе **Main** с помощью методов **Write** и **ReadLine** запросите у пользователя значения операндов и символа операции:

```
Console.Write("A = ");
double a = double.Parse(Console.ReadLine());
Console.Write("OP = ");
char op = char.Parse(Console.ReadLine());
Console.Write("B = ");
double b = double.Parse(Console.ReadLine());
```

- Объявите и присвойте начальные значения булевой переменной (она будет использоваться при проверке символа операции) и вещественной переменной (результата операции):

```
bool ok = true;
double res = 0;
```

➢ **Реализуйте работу оператора switch**

- Реализуйте логику работы переключателя, при этом учтите, что операция деления возможна с помощью двух символов “/” и “**:**”

```
switch (op)
{
   case '+': res = a + b; break;
   case '-': res = a - b; break;
   case '*': res = a * b; break;
   case '/':
   case ':':
             res = a / b; break;
   default: ok = false; break;
}
```

- Перед выводом результата проверьте переменную **ok**, и в случае ее значения **false** выведете сообщение об ошибке выбора типа операции:

```
if (ok) Console.WriteLine("{0} {1} {2} = {3}", a, op, b,
res);
    else Console.WriteLine("Операция не определена");
```

➢ **Протестируйте работу программы**

- Постройте и запустите приложение.
- Протестируйте работу программы с правильными значениями.
- Проверьте работу при делении на нуль, делении нуль на нуль. Объясните полученные результаты.
- Проверьте работу при неправильном указании символа операции, объясните полученный результат.

Задание 3. Определение високосного года

Дано натуральное число. Требуется определить, является ли год с данным номером високосным. Если год является високосным, то выведите YES, иначе выведите NO.

Указание: год является високосным, если его номер кратен 4, но не кратен 100, а также, если он кратен 400.

Упражнение 2. Реализация циклов при работе с данными размерных типов

Задание 1. Использование операторов цикла *while, do while* и *for*.

В этом задании вы напишите программу, выводящую на экран последовательность целых нечетных чисел в строчку через пробел с помощью трех операторов цикла **while**, **do while** и **for**.

➢ **Создайте проект**

- Создайте проект Loop.sln в папке *install folder*\Labs\Lab03\.

➢ **Реализуйте запрос количества чисел**

- В методе **Main** с помощью методов **Write** и **ReadLine** запросите у пользователя количество чисел:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
```

➢ **Реализуйте вывод чисел с помощью операторов цикла**

- Реализуйте вывод последовательности целых нечетных чисел в строчку через пробел с помощью оператора цикла

// while:

```
Console.Write("\nwhile: \t\t");
int i = 1;
while (i <= n)
   {
      Console.Write(" " + i);
      i += 2;
   }
```

// do while:

```
Console.Write("\ndo while: \t");
```

```
i = 1;
do
   {
      Console.Write(" " + i);
      i += 2;
   }
while (i <= n);
```

// **for:**

```
Console.Write("\nFor: \t\t");
for (i = 1; i<=n; i+=2)
    {
      Console.Write(" " + i);
    }
```

- Постройте и протестируйте приложение.

➢ **Используйте цикл с постусловием**

- В методе `main()` объявите четыре переменных вещественного типа, `x` – аргумент функции, `x1`, `x2` – границы интервала, `y` – выходной параметр функции, для границ интервала реализуйте ввод значений с клавиатуры.
- Реализуйте печать заголовка таблицы вывода значений функции.
- С помощью цикла с постусловием реализуйте вывод значений функции `sin(x)` на интервале от `x1` до `x2` с шагом 0,01:

```
x = x1;
do
{
     y = sin(x);
     реализация вывода
     x = x + 0.01;
}
while (x <= x2);
return 0;
}
```

- Постройте и протестируйте приложение.

➢ **Используйте цикл с предусловием**

- В функции `main()` объявите три целочисленные переменные, `a` и `b` – исходные данные, `temp` – временная переменная для реализации алгоритма.
- Реализуйте ввод значений переменных `a` и `b`.
- С помощью цикла с предусловием реализуйте алгоритм Евклида:

```
temp = a;
while (temp!=b)
{
     a = temp;
     if (a<b)
     {
          temp = a;
```

```
            a = b;
            b = temp;
        }
        temp = a - b;
        a = b;
    }
```

- Постройте и протестируйте приложение.

➢ **Сравните типы цикла**

- Реализуйте задачу вывода значений функции с помощью цикла с предусловием, а алгоритм Евклида с постусловием.
- Сравните варианты реализации задач с помощью разных циклов и сделайте выводы о целесообразности выбора типа цикла.

Задание 2. Расчет суммы, используя операторы перехода

В этом задании составьте программу, реализующую сумму:

$$s = \sum_{i=1}^{100} i,$$

для i, находящихся от 1 до *k* и от *m* до 100.

Указание. Для суммирования чисел в диапазоне можно воспользоваться циклом **for** и оператором перехода **continue**:

```
for (i=1;i<=100;i++)
  {
   if (i>k && i<m) continue;
   s+=i;
  }
```

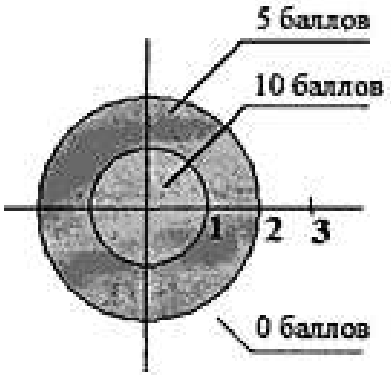
Задание 3. Стрельба по мишени

В этом задании разработайте программу, имитирующую стрельбу по мишени.

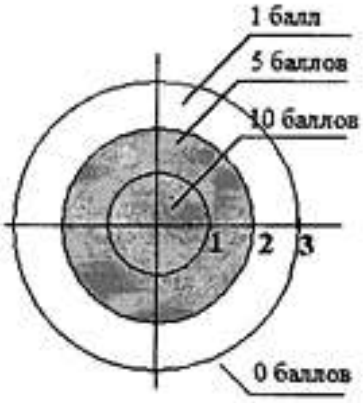
Реализуйте следующую функциональность:

- Пользователь вводит данные о выстреле в виде пары чисел – координат `x` и `y` несколько раз.
- Повтор ввода следует организовать в цикле. После «стрельбы» пользователю выводится информация о сумме очков.

Вариант мишени определяется номером обучающегося по списку: нечетный – вариант 1, четный – вариант 2.



Вариант 1.



Вариант 2.

Дополнительные указания:

- реализовать центр мишени случайным значением, тогда стрелок не будет знать местонахождение мишени (стрельба вслепую);
- реализовать случайную помеху при выстреле, тогда стрелок будет использовать трудности при стрельбе.

Лабораторная работа 4. Создание и использование методов

Цель работы

Изучение и приобретение навыков работы с методами класса

Упражнение 1. Использование параметров в методах, возвращающих значения

В этом упражнении вы создадите класс **Utils,** в котором определите метод **Greater.** Этот метод будет принимать два целочисленных параметра и возвращать больший из них. Для тестирования работы данного класса Вы будете использовать класс **Program,** в котором у пользователя будут запрашиваться два числа, далее будет вызываться метод **Utils.Greater**, после чего на экран консоли будет выводиться результат.

➢ **Создайте проект**

- Создайте проект Utils.sln в папке *install folder*\Labs\Lab04\.
- Добавьте в проект новый класс с именем **Utils**: **Projects** (Проект) → **Add class** (Добавить класс).

➢ **Создайте метод Greater класса Utils**

- Создайте статический метод **Greater** следующим образом:
  - Откройте класс **Utils.**
  - В класс **Utils** добавьте статический открытый метод **Greater.** Этот метод должен использовать два передаваемых *по значению* параметра *a* и *b* типа **int** и возвращать значение типа **int**, являющееся большим из двух передаваемых значений**:**

```
class Utils
{
    public static int Greater(int a, int b)
    {
        if (a > b)
            return a;
        else
            return b;
    }

}
```

- **Протестируйте метод Greater**

  - Откройте класс **Program**.
  - Внутри метода **Main** реализуйте следующее:
    - Объявите две целочисленные переменные *x* и *y*. Добавьте код для их считывания с клавиатуры (используйте методы **Console.ReadLine** и **int.Parse**):

```
int x;
int y;

Console.WriteLine("Введите первое число:");
x = int.Parse(Console.ReadLine());
Console.WriteLine("Введите второе число:");
y = int.Parse(Console.ReadLine());
```

    - Протестируйте метод **Greater,** вызвав его на исполнение от имени класса (так как метод статический) и присвоив возвращенное им значение целочисленной переменной *greater*:

```
int greater = Utils.Greater(x, y);
```

    - Напишите код, выводящий на консоль большее из двух чисел, используя метод **Console.WriteLine:**

```
Console.WriteLine("Большим из чисел {0} и {1} является {2} ", x , y , greater);
```

  - Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 2. Использование в методах параметров, передаваемых по ссылке

В этом упражнении вы создадите метод **Swap**, который поменяет местами значения параметров. При этом вы будете использовать параметры, передаваемые по ссылке.

- **Создайте метод Swap**

  - Добавьте в класс **Utils** статический открытый метод **Swap,** который должен будет принимать два параметра *a* и *b,* передаваемых по ссылке:

```
public static void Swap(ref int a, ref int b)
{
```

- Внутри метода **Swap** напишите код, меняющий местами значения *a* и *b*. Необходимо будет создать дополнительную локальную переменную *temp* типа **int**, в которой в процессе перестановки значений *a* и *b,* будет временно храниться значение одной из переменных:

```
int temp = a;
a = b;
b = temp;
```

- Постройте приложение.

➢ **Протестируйте метод Swap**

- В методе **Main** класса **Program** вызовите метод **Swap**, передав по ссылке значения переменных *x* и *y* в качестве параметров:

```
Utils.Swap(ref x, ref y);
```

- Выведите на экран значения переменных *x* и *y* до и после перестановки:

```
Console.WriteLine("До swap: \t" + x + " " + y);
Utils.Swap(ref x, ref y);
Console.WriteLine("После swap: \t" + x + " " + y);
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

*Замечание***:** Если значения переменных не поменялись местами, удостоверьтесь, что вы передавали параметры по ссылке (с модификатором **ref).**

Упражнение 3. Использование возвращаемых параметров в методах

В этом упражнении вы создадите метод **Factorial**, принимающий целочисленную переменную и рассчитывающий ее факториал по итерационному алгоритму.

➢ **Создайте метод Factorial**

- Добавьте в класс **Utils** статический открытый метод **Factorial,** имеющий следующие особенности реализации:
  - Метод **Factorial** будет использовать два параметра *n* и *answer***:**
    - первый параметр типа **int** передается по значению (это число, для которого рассчитывается факториал),
    - второй параметр типа **out int** используется для возвращения результата.
  - Метод **Factorial** должен возвращать значение типа **bool**, отражающее успешность выполнения метода, так как может произойти переполнение и выброс исключения.

```
public static bool Factorial(int n, out int answer)
{

}
```

- Внутри метода напишите код расчета факториала для передаваемого на вход значения:
  - Объявите три переменных: целочисленные *k* (будет использоваться в цикле в качестве счетчика) и *f* (будет использоваться внутри цикла, присвойте начальное значение 1), а также булевого типа *ok* (будет использоваться для отслеживания ошибок):

```
int k;           // Loop counter
int f = 1;       // Working value
bool ok = true; // True if okay, false if not
```

  - Создайте цикл **for**. Начальное значение *k* = 2, итерации продолжаются до тех пор, пока не будет достигнуто значение параметра *n*. На каждом шаге значение *k* увеличивается на единицу. В теле цикла *f* умножается на *k* и сохраняется результат в *f*. Значение факториала растет достаточно быстро, поэтому реализуйте проверку на арифметическое переполнение в блоке **checked**:

```
checked
{
  for (k = 2; k <= n; ++k)
    {
        f = f * k;
    }
}
```

  - Реализуйте перехват возможных исключений, все исключения обрабатываются одинаково: обнуляется результат и переменной *ok* присваивается **false**:

```
try
{
    checked
    {
      for (k = 2; k <= n; ++k)
        {
            f = f * k;
        }
    }
}
catch (Exception)
{
    f = 0;
    ok = false;
}
```

  - Итоговое значение переменной *f* присвойте возвращаемому параметру *answer*:

```
answer = f;
return ok;
```

Если метод отработал успешно, он возвращает значение **true**, если произошло арифметическое переполнение (выброс исключения), то возвращается значение **false**.

➢ **Протестируйте метод Factorial**

- Отредактируйте класс **Program**, выполнив следующее:
  - Объявите переменную *ok* типа **bool** для хранения возвращаемого методом значения (**true** или **false**) и переменную *f* типа **int** для хранения факториала числа, рассчитанного в методе:

```
int f;
bool ok;
```

  - Запросите у пользователя целое число и сохраните введенное значение в переменной *x* типа **int**:

```
Console.WriteLine("Number for factorial:");
x = int.Parse(Console.ReadLine());
```

  - Вызовите метод **Factorial,** передав *x* и *f* в качестве параметров и возвращаемое методом значение присвойте переменной *ok*:

```
// Test the factorial function
ok = Utils.Factorial(x, out f);
```

  - Если переменная *ok* принимает значение **true**, выведите на консоль значение *x* и *f*, в противном случае выведите на экран сообщение об ошибке:

```
// Output factorial results
    if (ok)
Console.WriteLine("Factorial(" + x + ") = " + f);
else
Console.WriteLine("Cannot compute this factorial");
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 4. Расчет площади треугольника с помощью метода

Требуется создать класс **Operation,** в котором определите:

- статический метод расчета площади треугольника по формуле Герона (см. упражнение 1 лабораторной работы 2). Этот метод должен принимать три параметра (стороны треугольника) и возвращать значение площади;
- статический закрытый метод проверки наличия треугольника. Этот метод должен возвращать значение логического типа;
- перегруженный статический метод, который будет принимать один параметр – сторону и вычислять площадь равностороннего треугольника.

Указание. Вызов метода проверки наличия треугольника реализуйте в методе расчета площади.

Для тестирования работы данного класса используйте класс **Program,** в котором у пользователя сначала будет запрашиваться тип треугольника

(равносторонний или нет), а далее либо значение одной стороны, либо трех. После расчета площади на экран консоли будет выводиться результат.

Упражнение 5. Вычисление корней квадратного уравнения

Требуется реализовать метод вычисления корней квадратного уравнения.

- функция должна возвращать значение 1, если корни найдены, значение нуля, если оба корня совпадают, и значение -1, если корней не существует.
- значения корней уравнений должны возвращаться в качестве аргументов функции, передаваемых по ссылке.

Для тестирования работы данного класса используйте класс **Program,** в котором у пользователя будут запрашиваться три числа – коэффициенты уравнения, далее будет вызываться метод расчета корней уравнения, после чего на экран консоли будет выводиться результат (в зависимости от условий задачи) в следующем виде:

Корней уравнения с коэффициентами a = результат, b = результат, c = результат нет.

Корень уравнения с коэффициентами a = результат, b = результат, c = результат равны один x1= x2 = результат

Корни уравнения с коэффициентами a = результат, b = результат, c = результат равны x1 = результат, x2 = результат.

Лабораторная работа 5. Создание и использование массивов

Цель работы

Изучение массивов и приобретение навыков работы с ними.

Упражнение 1. Работа с массивом размерного типа данных

В этом упражнении вы реализуете в проекте **Loop** (лабораторная работа 3, упражнение 2) массив для хранения данных.

Откройте проект **Loop** (лабораторная работа 3, упражнение 2).

➢ **Добавьте возможность работы с массивом**

- В методе **Main** создайте массив:
```
int[] myArray = { 100, 1, 32, 3, 14, 25, 6, 17, 8, 99 };
```
- Реализуйте вывод элементов массива в строчку через пробел с помощью любого оператора цикла. При этом вместо четных элементов должно отображаться нули. Изменение четных чисел можно выполнить с помощью следующей проверки:
```
if (myArray[i] % 2 == 0) myArray[i] = 0;
```
- Постройте и протестируйте приложение.

➢ **Реализация определения пользователем верхней границы массива**

В этой части упражнения вы замените массив с явной инициализацией на массив, размер которого вводит с клавиатуры пользователь.

- Вместо создания массива с явной инициализацией сначала просто объявите массив:

```
int[] MyArray;
```

- Затем запросите у пользователя его размер и создайте массив требуемого размера:

```
int n = int.Parse(Console.ReadLine());
MyArray = new int[n];
```

- Реализуйте инициализацию элементов массива в цикле и при определении значения верхней границы используйте свойство `Length`:

```
for (int i = 0; i < MyArray.Length; ++i)
{
  Console.Write("a[{0}]=",i);
  MyArray[i] = int.Parse(Console.ReadLine());
}
```

- Для вывода на экран элементов массива используйте цикл **foreach:**

```
foreach (int x in MyArray)Console.Write("{0} ", x);
```

- Остальную функциональность упражнения оставьте без изменений. Постройте и протестируйте приложение.

Упражнение 2. Перемножение матриц

В этом упражнении вы напишите программу, в которой массивы будут использоваться для перемножения матриц. Программа будет считывать с консоли 4 целых числа и сохранять их в матрице размером 2x2, затем будут считываться еще 4 целых числа и сохраняться еще в одной матрице размером 2x2. Далее эти матрицы будут перемножаться, а результат сохранится в третьей матрице тех же размеров. Результат перемножения матриц выведется на экран консоли.

Для справки формула расчета произведения матриц A и B:

$$\begin{matrix} A1 & A2 \\ A3 & A4 \end{matrix} \times \begin{matrix} B1 & B2 \\ B3 & B4 \end{matrix} = \begin{matrix} A1.B1 + A2.B3 & A1.B2 + A2.B4 \\ A3.B1 + A4.B3 & A3.B2 + A4.B4 \end{matrix}$$

➢ **Создайте проект**

- Создайте проект MatrixMultiply.sln в папке *install folder*\Labs\Lab05\.
- Переименуйте файл и класс **Program** на **MatrixMultiply**.

➢ **Проверьте корректность операции перемножения двух матриц**

- В методе **Main** класса **MatrixMultiply** объявите массив целых чисел **a** размером 2x2. Позднее программа будет заполнять его числами, считанными с консоли.

- Пока заполните его значениями из рисунка, приведенного ниже. Это позволит проверить, что перемножение матриц реализовано корректно.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
int[,] a = new int[2, 2];
a[0, 0] = 1; a[0, 1] = 2;
a[1, 0] = 3; a[1, 1] = 4;
```

- Аналогичным образом добавьте в метод **Main** объявление еще одного массива целых чисел размером 2x2, назовите его **b**. Позднее программа также будет заполнять его числами, считанными с консоли, но пока заполните его значениями из рисунка:

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

- В методе **Main** объявите еще один массив целых чисел размером 2x2 и назовите его **result**. Заполните его значениями, рассчитанными по следующим формулам:

$$\begin{matrix} a[0,0] * b[0,0] + a[0,1] * b[1,0] & a[0,0] * b[0,1] + a[0,1] * b[1,1] \\ a[1,0] * b[0,0] + a[1,1] * b[1,0] & a[1,0] * b[0,1] + a[1,1] * b[1,1] \end{matrix}$$

- В метод **Main** добавьте 4 инструкции, выводящие на консоль значения массива **result**. Это поможет вам убедиться в том, что перемножение матриц выполнено корректно.
- Итоговый текст метода **Main** выглядит следующий образом:

```
static void Main( )
{
    int[,] a = new int[2,2];
    a[0,0] = 1; a[0,1] = 2;
    a[1,0] = 3; a[1,1] = 4;

    int[,] b = new int[2,2];
    b[0,0] = 5; b[0,1] = 6;
    b[1,0] = 7; b[1,1] = 8;

    int[,] result = new int[2,2];
    result[0,0]=a[0,0]*b[0,0] + a[0,1]*b[1,0];
    result[0,1]=a[0,0]*b[0,1] + a[0,1]*b[1,1];
    result[1,0]=a[1,0]*b[0,0] + a[1,1]*b[1,0];
    result[1,1]=a[1,0]*b[0,1] + a[1,1]*b[1,1];

    Console.WriteLine(result[0,0]);
    Console.WriteLine(result[0,1]);
    Console.WriteLine(result[1,0]);
    Console.WriteLine(result[1,1]);
}
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что на экран выводятся следующие значения массива **result:**

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

➢ **Выведите значения массива result, используя метод с параметром-массивом**

- С помощью механизма рефакторинга извлеките метод для вывода результирующего массива. Для этого выделите четыре инструкции, выводящие на консоль значения массива **result** Console.WriteLine(result[0,0]) и т.д. и в контекстном меню выберите команду **Рефакторинг → Извлечь метод**.
- В окне диалога **Извлечение метода** укажите имя нового метода **Output** и нажмите кнопку **ОК.**
- Проверьте, что в классе **MatrixMultiply** появился новый статический метод **Output**. Этот метод не возвращает значений и принимает в качестве параметра массив целых чисел второго ранга с именем **result**.
- Проверьте, что в методе **Main** добавился вызов метода **Output,** принимающий в качестве аргумента массив **result.**
- Запустите программу. Убедитесь в том, что на экран выводятся те же четыре значения.
- Измените метод **Output,** используя вместо четырех инструкций **Console.WriteLine** два вложенных цикла **for**. Для проверки верхних границ обоих массивов используйте вызов метода **GetLength**.

Итоговый текст метода **Output** должен выглядеть следующий образом:

```
private static void Output(int[,] result)
{
    for (int r = 0; r < result.GetLength(0); r++)
    {
     for (int c = 0; c < result.GetLength(1); c++)
        {
            Console.Write("{0} ", result[r, c]);
        }
        Console.WriteLine();
    }
}
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что на экран выводятся те же четыре значения.

- **Создайте метод, рассчитывающий и возвращающий значения массива result**
  - С помощью механизма рефакторинга извлеките метод для перемножения двух матриц. Для этого выделите объявление и инициализацию массива **result** из метода **Main,** в контекстном меню выберите команду **Рефакторинг → Извлечь метод** и в окне диалога **Извлечение метода** укажите имя нового метода **Multiply.**
  - Проверьте, что в классе **MatrixMultiply** объявлен новый статический метод **Multiply**. Этот метод возвращает массив целых чисел второго ранга и принимает в качестве параметров два массива целых чисел второго ранга с именами **a** и **b**.
  - Проверьте, что в методе **Main** добавилась инициализация массива **result** с вызовом метода **Multiply,** принимающим в качестве аргументов **a** и **b:**

```
int[,] result = Multiply(a, b);
```

Итоговый текст метода **Multiply** должен выглядеть следующий образом:

```
private static int[,] Multiply(int[,] a, int[,] b)
{
    int[,] result = new int[2, 2];
result[0, 0] = a[0, 0] * b[0, 0] + a[0, 1] * b[1, 0];
result[0, 1] = a[0, 0] * b[0, 1] + a[0, 1] * b[1, 1];
result[1, 0] = a[1, 0] * b[0, 0] + a[1, 1] * b[1, 0];
result[1, 1] = a[1, 0] * b[0, 1] + a[1, 1] * b[1, 1];
    return result;
}
```

  - Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что на экран выводятся те же четыре значения.

- **Рассчитайте значения массива result, используя цикл for**
  - В методе **Multiply** удалите инструкции инициализации массива **result**.
  - В метод **Multiply** добавьте два вложенных цикла **for**. Во внешнем цикле для итерации по каждому индексу первого измерения массива **result** используйте переменную *r* типа **int**. Во внутреннем цикле для итерации по каждому индексу второго измерения массива **result** используйте переменную *c* типа **int**. Для проверки верхних границ обоих массивов используйте вызов метода **GetLength**. В теле внутреннего цикла **for** необходимо рассчитать значение `result[r,c]` по следующей формуле:

```
result[r,c] = a[r,0] * b[0,c] + a[r,1] * b[1,c]
```

В итоге вместо четырех инструкций инициализации массива **result** должны быть два цикла **for**:

```
for (int r = 0; r < result.GetLength(0); r++)
{
    for (int c = 0; c < result.GetLength(1); c++)
```

```
            {
              result[r, c] += a[r, 0] * b[0, c] + a[r, 1] *
b[1, c];
            }
        }
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что на экран выводятся те же четыре значения.

➢ **Обеспечьте возможность считывания значений первой матрицы с консоли**

- С помощью механизма рефакторинга извлеките метод для ввода первой матрицы. Для этого после объявления матрицы **a** выделите четыре инструкции инициализации ее элементов, в контекстном меню выберите команду **Рефакторинг → Извлечь метод** и в окне диалога **Извлечение метода** укажите имя нового метода **Input.**
- Проверьте, что в классе **MatrixMultiply** объявлен новый статический метод **Input.** В этот метод просто были перенесены инструкции инициализации матрицы.
- Замените инструкции инициализации элементов массива **a** на два вложенных цикла **for**. Для проверки верхней границы массива используйте вызов метода **GetLength**. Во внутреннем цикле **for** для запроса у пользователя значений используйте инструкцию **Write**.

Итоговый текст метода **Input** должен выглядеть следующий образом:

```
private static void Input(int[,] a)
{
    for (int r = 0; r < a.GetLength(0); r++)
    {
        for (int c = 0; c < a.GetLength(1); c++)
        {
  Console.Write("Enter value for [{0},{1}] : ", r, c);
            string s = System.Console.ReadLine();
            a[r, c] = int.Parse(s);
        }
    }
    Console.WriteLine();
}
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Для заполнения массива **a** введите значения 1, 2, 3, 4. Убедитесь в том, что на экран выводятся те же четыре значения.

➢ **Обеспечьте возможность считывания с консоли значений второй матрицы**

- В методе **Main** удалите инструкции инициализации массива **b** и для считывания значений массива **b** добавьте вызов метода **Input**, передав в качестве аргумента массив **b**:

```
Input(b);
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Для заполнения массива **a** введите значения 1, 2, 3, 4. Для заполнения массива **b** введите значения 5, 6, 7, 8. Убедитесь в том, что на экран выводятся те же четыре значения: 19, 22, 43 и 50.
- Попробуйте вводить другие данные. Сравните свои результаты с результатами других обучаемых при вводе одинаковых данных.

Упражнение 3. Обработка данных массива

Требуется создать массив, заполнить его числами и выполнить обработку данных.

- С помощью цикла (тип цикла на ваше усмотрение) реализуйте заполнение массива с клавиатуры.
- Добавьте в программу **методы** для обработки данных массива:
    a) определение суммы всех элементов массива.
    b) среднего значения массива;
    c) расчет суммы отрицательных или положительных элементов,
    d) расчет суммы элементов с нечетными или четными номерами.
- Дополнительные задания – реализуйте методы, позволяющие:
    e) найти максимальный или минимальный элементы массива и вывести их индексы,
    f) рассчитать произведение элементов массива, расположенных между максимальным и минимальным элементами
- Постройте и протестируйте приложение.

Лабораторная работа 6. Создание и использование классов

Цель работы

Изучение понятия класса как пользовательского типа данных и приобретение навыков работы с классами.

Упражнение 1. Разработка класса Book

В этом упражнении вы создадите класс **Book** с соответствующими полями (автор или авторы, название, год издания и стоимость аренды за книгу) и методами.

➢ **Создайте проект**

- Создайте проект MyClass.sln в папке *install folder*\Labs\Lab06\.
    - Добавьте в проект новый класс с именем **Book**: **Projects** (Проект) → **Add class** (Добавить класс).

- **Реализуйте класс Book**
  - Добавьте в класс закрытые поля:
```
private String author;         // автор
private String title;          // название
private String publisher; // издательство
private int pages;             // кол-во страниц
private int year;              // год издания
```
  - Добавьте в класс статическое поле, которое определяет стоимость аренды за любую книгу:
```
private static double price = 9;
```
  - Реализуйте метод, устанавливающий значения характеристик книги:
```
public void SetBook(String author, String title,
String publisher, int pages, int year)
{
   this.author = author;
   this.title = title;
   this.publisher = publisher;
   this.pages = pages;
   this.year = year;
}
```
  - Реализуйте статический метод устанавливающий стоимость аренды, так как статическое поле будет существовать в единственном экземпляре для всех объектов класса, обращаются к нему не через имя экземпляра, а через имя класса:
```
public static void SetPrice(double price)
{
   Book.price = price;
}
```
  - Реализуйте метод, выводящий на экран информацию о книге:
```
public void Show()
{
Console.WriteLine("\nКнига:\n Автор: {0}\n Название:
{1}\n Год издания: {2}\n {3} стр.\n Стоимость аренды:
{4}", author, title, year, pages, Book.price);
 }
```
  - Реализуйте метод, определяющий стоимость аренды за указанное количество суток:
```
public double PriceBook(int s)
{
     double cust = s * price;
     return cust;
}
```

- **Протестируйте работу класса Book**
  - Откройте класс **Program.**
  - В методе **Main** создайте отрезок – объект класса **Book:**
```
Book b1 = new Book();
```
  - Вызовите метод для инициализации книги:

```
    b1.SetBook("Пушкин А.С.", "Капитанская дочка",
"Вильямс", 123, 2012);
```

- С помощью статического метода установите стоимость аренды книги:
```
    Book.SetPrice(12);
```
- Вызовите метод для вывода информации о книге:
```
    b1.Show();
```
- Отобразите на экране стоимость аренды за книгу в течении трех суток:
```
    Console.WriteLine("\n Итоговая стоимость аренды: {0}
p.", b1.PriceBook(3));
```
- Постройте проект и исправьте ошибки, если это необходимо. Запустите программу. Сравните реализацию и использование статического метода и метода экземпляра.

Упражнение 2. Использование конструкторов

В этом упражнении вы добавите конструкторы для инициализации объекта. Конструктор экземпляра вызывается автоматически при создании объекта класса с помощью операции **new**, причем имя конструктора совпадает с именем класса. Также вы будете использовать статический конструктор для инициализации статического поля класса.

➢ **Добавьте конструктор экземпляра для инициализации полей объекта**

- Откройте файл Book.cs.
- Добавьте в класс **Book** конструктор, который инициализирует все поля объекта:
```
    public Book(String author, String title, String
publisher, int pages, int year)
    {
       this.author = author;
       this.title = title;
       this.publisher = publisher;
       this.pages = pages;
       this.year = year;
    }
```
- Проверьте, что в окне ошибок появилось сообщение об ошибке: класс не содержит конструктор, который принимает аргументы “0”. Это объясняется тем, что попытка объявления любого варианта конструктора приводит к тому, что транслятор перестает заниматься построением собственных версий конструкторов и вызов конструктора по умолчанию (`Book b1 = new Book();`) приведет к ошибке.
- Добавьте конструктор по умолчанию:
```
            public Book ()
            { }
```

- В методе **Main** создайте новую книгу, при этом будет вызван конструктор с параметрами:
```
Book b2 = new Book("Толстой Л.Н.", "Война и мир",
"Наука и жизнь", 1234, 2013);
```
- Вызовите метод для вывода информации о новой книге:
```
b2.Show();
```
- Постройте проект и исправьте ошибки, если это необходимо. Запустите программу. Обратите внимание на значение поля “Стоимость аренды в сутки”.

➢ **Добавьте статический конструктор**

Статический конструктор автоматически вызывается системой до первого обращения к любому элементу класса, выполняя необходимые действия по инициализации.

- Добавьте в класс **Book** статический конструктор, который инициализирует статическое поле класса:
```
static Book()        //статический конструктор
{
   price = 10;
}
```
- Постройте проект и исправьте ошибки, если это необходимо. Запустите программу. Обратите внимание на значение поля “Стоимость аренды в сутки”.

➢ **Используйте перегрузку конструкторов**

- Можно задать в классе несколько конструкторов, чтобы обеспечить возможность инициализации объектов разными способами. Добавьте в класс **Book** конструктор, который задает только автора и название, а остальные поля получат свои значения благодаря конструктору по умолчанию:
```
public Book(String author, String title)
{
     this.author = author;
     this.title = title;
 }
```
- В методе **Main** создайте объект, передав в качестве параметров значения автора и названия книги:
```
Book b3 = new Book("Лермонтов М.Ю.", "Мцыри");
```
- Проверьте информацию о новой книге:
```
b3.Show();
```
- Постройте проект и исправьте ошибки, если это необходимо. Запустите программу.

Упражнение 3. Реализация класса Triangle

В этом упражнении требуется создать класс **Triangle**, разработав следующие элементы класса:

- Поля: стороны треугольника.
- Конструктор, позволяющий создать экземпляр класса с заданными длинами сторон.
- Методы, позволяющие:
  - вывести длины сторон треугольника на экран;
  - расчитать периметр треугольника;
  - расчитать площадь треугольника;р
  - реализовать проверку, позволяющую установить, существует ли треугольник с данными длинами сторон.

Лабораторная работа 7. Создание иерархии классов

Цель работы

Изучение наследования как важного элемента объектно-ориентированного программирования и приобретение навыков реализации иерархии классов.

Упражнение 1. Реализация наследования классов

В этом упражнении вы будете использовать наследование для построения иерархии между классами, имеющих отношение типа "**является**". Вы добавите новый класс **Item**, который будет являться базовым для уже имеющегося класса **Book**.

Предположим, что для библиотечной системы, которую требуется разработать, необходимо создать классы, описывающие различные книги, журналы и т.п., которые хранятся в библиотеке.

Книга, журнал, газета обладают как общими, так и различными свойствами. Например, у книги имеется автор или авторы, название и год издания, у журнала есть название, номер и содержание – список статей. В то же время книги, журналы и т.д. имеют и общие свойства: все это – "единицы хранения" в библиотеке, у них есть инвентарный номер, они могут быть в читальном зале, у читателей или в фонде хранения. Их можно выдать и, соответственно, сдать в библиотеку. Эти общие свойства удобно объединить в одном базовом классе. В этом упражнении вы разработаете класс **Item**, который описывает единицу хранения в библиотеке.

➢ **Выполните подготовительные операции**

- Создайте папку Lab07 и скопируйте в нее решение MyClass, созданное в прошлом упражнении.

➢ **Создайте базовый класс Item**

- Откройте проект MyClass.sln в папке *install folder*\Labs\Lab07\.
- Добавьте в проект новый класс с именем **Item**: **Projects** (Проект) → **Add class** (Добавить класс).

- Добавьте в класс защищенные поля, соответствующие единице хранения с помощью спецификатора доступа **protected**. Этот спецификатор обеспечивает открытый доступ к членам базового класса, но только для производного класса:

```
// инвентарный номер — целое число
protected long invNumber;
// хранит состояние объекта - взят ли на руки
protected bool taken;
```

- Добавьте методы базового класса, реализующие работу с единицей хранения:

```
// истина, если этот предмет имеется в библиотеке
public bool IsAvailable()
      {
          if (taken == true)
              return true;
          else
              return false;
      }
// инвентарный номер
   public long GetInvNumber()
      {
          return invNumber;
      }

// операция "взять"
   public void Take()
        {
            taken = false;
        }

// операция "вернуть"
   public void Return()
       {
           taken = true;
       }
```

- Реализуйте метод, выводящий на экран информацию о единице хранения:

```
public void Show()
{
  Console.WriteLine("Состояние единицы хранения:\n
Инвентарный номер: {0}\n Наличие: {1}", invNumber,
taken);
}
```

➢ **Реализуйте отношение наследования**

- После имени класса **Book** укажите имя базового класса (класс **Book** теперь производный от класса **Item)** и удалите объявление полей начала отрезка (координаты x и y):

```
class Book : Item
{
. . .
```

Производный класс наследует все переменные и методы, определенные в базовом классе. В производном классе ранее был реализован метод **Show()**, теперь он скрывает идентичный метод базового класса.

- Укажите ключевое слово **new** в определении метода **Show()** производного класса для явного указания факта скрытия метода базового класса:

```
new public void Show()
{
    Console.WriteLine("\nКнига:\n Автор: {0}\n
Название: {1}\n Год издания: {2}\n {3} стр.\n Стоимость
аренды: {4} р. в сутки", author, title, year, pages,
Book.price);
}
```

- Реализуйте процесс выдачи книг. Для этого достаточно тех возможностей, которые представляет базовый класс. Добавьте метод выдачи книги:

```
public void TakeItem()
{
    if (this.IsAvailable())
    this.Take();
}
```

Для возврата книг можно воспользоваться соответствующим методом базового класса.

➢ **Протестируйте отношения наследования**

- В методе **Main** класса **Program** создайте объект – единицу хранения с помощью конструктора по умолчанию базового класса и отобразите на экране ее значения по умолчанию:

```
Item item1 = new Item();
item1.Show();
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите программу.
- Заметьте, что информация о наличии выводится как **false**, что противоречит здравому смыслу, единицы хранения должны быть в наличии в библиотеке. Это противоречие будет устранено в следующем упражнении.
- Обратите внимание, что конструктор определяется только в производном классе, поэтому часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора по умолчанию, а часть объекта, соответствующая производному классу, создается собственным конструктором.

Упражнение 2. Использование конструкторов

В этом упражнении вы определите конструкторы в базовом и в производном классах и реализуете их выполнение в обоих классах с

помощью ключевого слова **base**, которое позволяет вызвать конструктор базового класса.

➢ **Добавьте конструкторы в базовый и в производный классы**

- Добавьте в базовый класс **Item** два конструктора:
  - первый конструктор с параметрами для инициализации объекта хранения при его создании:

```
public Item(long invNumber, bool taken)
{
     this.invNumber = invNumber;
     this.taken = taken;
}
```

  - второй конструктор по умолчанию для инициализации начальными значениями, учитывая, что в начале работы предполагается наличие элемента в библиотеке:

```
public Item()
{
    this.taken = true;
}
```

- Добавьте конструктор со ссылкой на конструктор базового класса:

```
    public Book(String author, String title, String
publisher, int pages, int year, long invNumber, bool
taken) : base (invNumber, taken)
    {
         this.author = author;
         this.title = title;
         this.publisher = publisher;
         this.pages = pages;
         this.year = year;
    }
```

➢ **Вызовите метод базового класса**

В производном классе метод **Show** скрывает соответствующий метод базового класса. Для вывода информации о тех полях, которые наследуются производным классом необходимо вызвать метод именно базового класса. Для этого применяется также ключевое слово **base**.

- В метод **Show** класса **Book** добавьте вызов метода **Show** базового класса:

```
    new public void Show()
    {
         Console.WriteLine("\nКнига:\n Автор: {0}\n
Название: {1}\n Год издания: {2}\n {3} стр.\n Стоимость
аренды: {4} р. в сутки", author, title, year, pages,
Book.price);
         base.Show();
    }
```

➢ **Протестируйте отношения наследования**

- В методе **Main** класса **Program** внесите изменения в код создания объекта b2– книгу с помощью конструктора базового класса:

```
    Book b2 = new Book("Толстой Л.Н.", "Война и мир",
"Наука и жизнь", 1234, 2013, 101, true);
```

- Вызовите метод выдачи книги и отобразите данные о ней:

```
    b2.TakeItem();
    b2.Show();
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.
- Обратите внимание, что при создании книги инициализируются поля, наследуемые от базового класса, а после выдачи книги отображается информация о ее отсутствии.

➢ **Добавьте класс Magazine производный от класса Item**

- Добавьте в проект новый класс с именем **Magazine**: **Projects** (Проект) → **Add class** (Добавить класс):

```
    class Magazine : Item
    {
    }
```

- Добавьте в класс переменные-поля для описывания журнала:

```
    private String volume;    // том
    private int number;        // номер
    private String title;       // название
    private int year;      // год выпуска
```

- Добавьте конструкторы с параметрами и по умолчанию:

```
    public Magazine(String volume, int number, String
title, int year, long invNumber, bool taken) :
base(invNumber, taken)
    {
        this.volume = volume;
        this.number = number;
        this.title = title;
        this.year = year;
    }
    public Magazine()
           { }
```

- Добавьте метод, отображающий информацию об объекте – журнале:

```
    new public void Show()
           {
       Console.WriteLine("\nЖурнал:\n Том: {0}\n Номер:
{1}\n Название: {2}\n {3} \n Год выпуска: {4}",volume,
number, title, year);
    base.Show();
           }
```

- Журналы, как и книги, тоже должны выдаваться. Поэтому перенесите метод **TakeItem** из класса **Book** в базовый класс **Item**. Теперь и объекты – журналы тоже получат к нему доступ.

- Теперь метод **Take** базового класса применяется только внутри класса (в методе **TakeItem**) и для закрытия доступа к нему из других классов укажите модификатор доступа для метода **Take** как **private.**

➢ **Протестируйте работу класса Magazine**

- В методе **Main** класса **Program** создайте объект класса **Magazine** – журнал с передачей параметров в конструктор:

```
Magazine mag1 = new Magazine("О природе", 5,"Земля и мы", 2014, 1235, true);
```

- Вызовите метод выдачи журнала и отобразите информацию о нем:

```
mag1.Show();
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

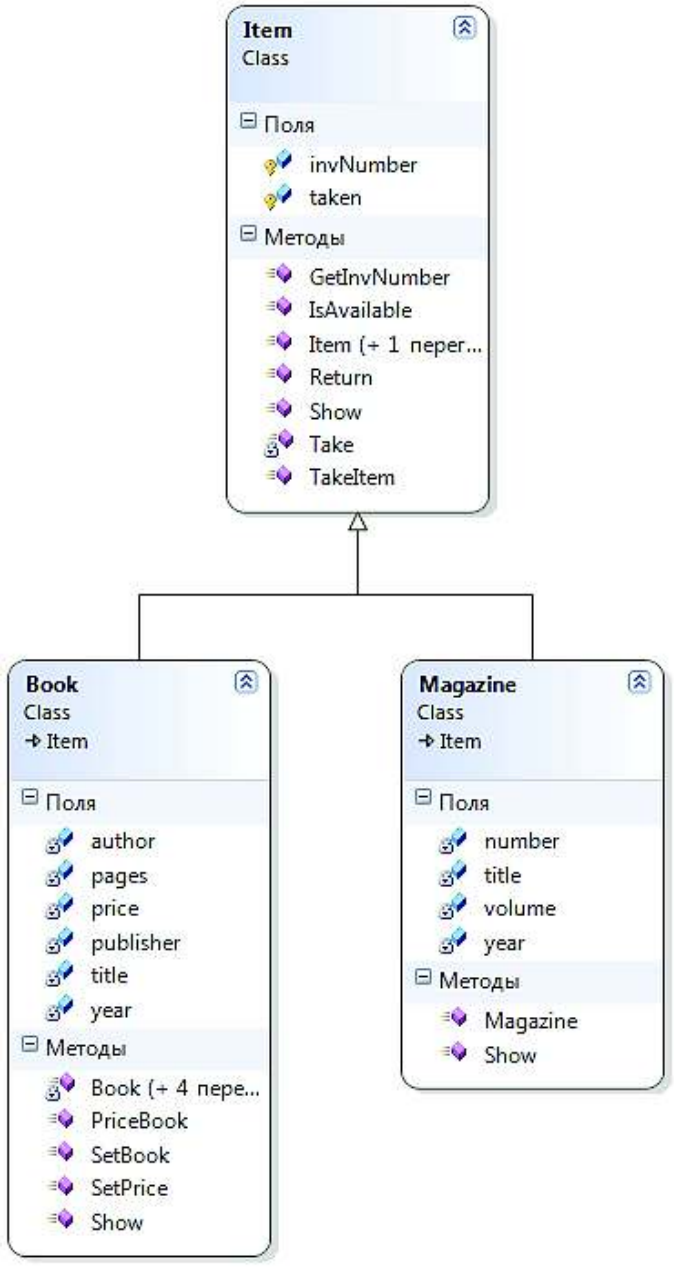


Рис. 7.2.1 Диаграмма классов

- Просмотрите диаграмму разработанных классов. Диаграмма классов должна выглядеть как на рис. 7.2.1.

Упражнение 3. Переопределение методов

В этом упражнении вы реализуете механизм полиморфизма с помощью виртуальных методов и их переопределения в производных классах.

➢ Определите, какой метод базового класса должен быть виртуальным

Процесс выдачи и приема изданий различается в классах **Book** и **Magazine**.

В первом классе предполагается, что берется плата за аренду книг, а во втором – журналы можно брать бесплатно. В связи с этим логично предположить, что важно при приеме книги убедиться в том, что книга возвращается вовремя и не требуется наложение штрафа за просроченный срок. Поэтому метод **Return** в производных классах будет разным, и есть смысл в базовым классе объявить его виртуальным.

- В классе **Item** укажите, что метод виртуальный:

```
public virtual void Return()    // операция "вернуть"
{
    taken = true;
}
```

- Для учета возвращения книги в класс **Book** добавьте новое поле булевого типа:

```
private bool returnSrok;
```

- Добавьте метод устанавливающий, что книга сдана в срок:

```
public void ReturnSrok()
{
      returnSrok = true;
}
```

- Переопределите виртуальный метод базового класса в производном классе с помощью ключевого слова **override:**

```
public override void Return()    // операция "вернуть"
{
     if (returnSrok == true)
          taken = true;
      else
          taken = false;
        }
```

- В классе **Magazine** определите метод **Return** с указанием ключевого слова **override**:

```
public override void Return()    // операция "вернуть"
{
   taken = true;
}
```

- Укажите в базовом классе метод **Show** как виртуальный и в производных классах переопределите его, добавив ключевое слово **override**.

Таким образом, каждый производный класс имеет свою собственную версию виртуального метода. Это используется тогда, когда доступ к объекту производного класса осуществляется через ссылочную переменную базового класса. В этой ситуации C# сам выбирает какую версию виртуального метода нужно вызвать. Этот выбор производится по типу объекта, на который ссылается данная ссылка.

- В методе **Main** класса **Program** объявите ссылку на объект базового класса:

```
Console.WriteLine("\n Тестирование полиморфизма");
Item it;
```

- Присвойте ссылке базового класса на объект производного класса – книга `b2` (этот объект был ранее создан) и от имени переменной базового класса вызовите виртуальные методы и так как методы виртуальны, то вызов приводит к вызову методов производного класса:

```
it = b2;
it.TakeItem();
it.Show();
```

- То же выполните и для объекта другого производного класса:

```
it = mag1;
it.TakeItem();
it.Show();
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 4. Применение абстрактного класса и абстрактных методов

В этом упражнении вы сделаете базовый класс и один из его методов абстрактным. Создавать объекты класса **Item** не имеет смысла, его назначение – быть базовым, и создавать экземпляры этого класса нельзя.

➢ Реализуйте класс Item как абстрактный

- Добавьте ключевое слово **abstract** при объявлении класса

```
abstract class Item
    {
```

Производные классы не обязаны переопределять виртуальные методы, но если требуется обязать переопределять виртуальный метод, то его следует сделать абстрактным. Методы, помеченные как **abstract**, являются чистым протоколом: они просто определяют имя, возвращаемый тип (если есть), и набор параметров (при необходимости).

➢ **Реализуйте метод Return как абстрактный**

- Добавьте ключевое слово **abstract** при объявлении метода **Return** базового класса и удалите тело класса:

```
abstract public  void Return();
```

➢ **Протестируйте вызов абстрактного метода**

- В методе **Main** класса **Program** вызовите абстрактный метод:

```
Item it;
it = b2;
it.TakeItem();
it.Return();
it.Show();
it = mag1;
it.TakeItem();
it.Return();
it.Show();
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 5. Реализации модели включения

Рассмотренное в предыдущих упражнениях наследование представляет собой отношение “является”. В этом упражнении вы реализуете отношение “имеет”, известное под названием модели включения или агрегации.

Агрегация – отношение типа “часть-целое” (“part of” – логическое включение). Эта форма повторного использования не применяется для установки отношений “родительский-дочерний”. Вместо этого такое отношение позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность (при необходимости) пользователю объекта.

➢ **Создайте проект**

- Создайте проект MyClassLine.sln в папке *install folder*\Labs\Lab07\.

➢ **Создайте класс Point**

- Добавьте в проект новый класс с именем **Point**: **Projects** (Проект) → **Add class** (Добавить класс).

➢ **Реализуйте поля и конструкторы класса**

- Добавьте в поля класса закрытые переменные – координаты точки:

```
class Point
{
    private double x;
    private double y;
```

- Добавьте конструктор с параметрами и конструктор по умолчанию:

```
public Point(double x, double y)
        {
        this.x = x;
        this.y = y;
        }

public Point ()
        { }
```

- **Реализуйте методы класса**
  - Добавьте метод, который отображает информацию об объекте – точке:

```
public void Show()
{
 Console.WriteLine("Точка с координатами: ({0}, {1})",
x, y);
}
```

  - Добавьте метод, определяющий расстояние между двумя точками, первая точка будет текущим объектом, а вторая передается в качестве параметра:

```
public double Dlina(Point p)
{
   double Dl = Math.Sqrt((this.x - p.x) * (this.x -
p.x) + (this.y - p.y) * (this.y - p.y));
   return Dl;
}
```

  - Переопределите метод **ToString** для отображения объекта:

```
public override string ToString()
{
     string ss =  x + " ; "+ y;
       return ss;
 }
```

- **Создайте класс Line**
  - Добавьте в проект новый класс с именем **Line**: **Projects** (Проект) → **Add class** (Добавить класс).

Между классами **Point** и **Line** отношение наследования вряд ли возможно, так как линия не “является” точкой. Между этими классами можно определить отношения агрегации – точка является частью линии и можно сказать, что отрезок “имеет” точки – начальную и конечную.

- **Реализуйте поля и конструкторы класса**
  - Добавьте в поля класса закрытые переменные – два объекта – точки начальную и конечную:

```
private Point pStart();
private Point pEnd();
```

  - Добавьте конструктор с параметрами и конструктор по умолчанию:

```
public Line(Point pStart, Point pEnd)
```

```
{
     this.pStart = pStart;
      this.pEnd = pEnd;
 }

public Line()
{ }
```

- **Реализуйте методы класса**
  - Добавьте метод, который отображает информацию об объекте – отрезке:

```
public void Show()
{
   Console.WriteLine("Отрезок с координатами: ({0}) -
({1})", pStart, pEnd);
}
```

- **Реализуйте функциональность включенного объекта**
  - Для представления функциональности включенного объекта реализуйте метод определения длины отрезка с помощью метода класса **Point**:

```
public double DlinL()
{
    return pStart.Dlina(pEnd);
}
```

- **Протестируйте модели включения**
  - В методе **Main** класса **Program** создайте две точки и отобразите информацию о них:

```
Point p1 = new Point();
p1.Show();
Point p2 = new Point(12, 13);
p2.Show();
```

  - Создайте отрезок и отобразите его координаты:

```
Line line = new Line(p1,p2);
line.Show();
```

  - Определите длину отрезка и отобразите на экран результат:

```
double dtr = line.DlinL();
Console.WriteLine("Длина отрезка " + dtr);
```

  - Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 6. Реализация отношения ассоциации между классами

Ассоциация – наиболее общий тип отношений, связанных с передачей сообщений ассоциированным классам путем делегирования. Реализуется с помощью включения ссылок на ассоциированные классы в список атрибутов класса.

В этом упражнении вы создадите класс, моделирующий игральную кость и класс игрока, который связан с классом игры отношением ассоциации – каждый отдельный сеанс игры связан с конкретным игроком, бросающим кубик.

➢ **Создайте проект**

- Создайте проект Igra.sln в папке *install folder*\Labs\Lab07\.

➢ **Создайте класс IgralnayaKost**

- Добавьте в проект новый класс с именем **IgralnayaKost**: **Projects** (Проект) → **Add class** (Добавить класс).

➢ **Реализуйте поля и конструктор класса**

- Добавьте в поля класса объект класса `Random`, предоставляющий генератор псевдослучайных чисел:

```
class IgralnayaKost
{
     Random r;
...
```

- Добавьте конструктор класса, в котором создается экземпляр класса `Random`:

```
public IgralnayaKost()
{
   r = new Random();
}
```

➢ **Реализуйте метод класса**

- Добавьте метод, который реализует случайное число в диапазоне от 1 до 6:

```
public int random()
{
     int res = r.Next(6)+1;
     return res;
}
```

➢ **В итоге класс может быть таким:**

```
class IgralnayaKost
{
     Random r;

     public IgralnayaKost()
     {
          r = new Random();
     }
     public int random()
     {
          int res = r.Next(6)+1;
```

```
            return res;
        }
    }
```

➢ **Создайте класс Gamer**

- Добавьте в проект новый класс с именем **Gamer**: **Projects** (Проект) → **Add class** (Добавить класс).

Между классами **IgralnayaKost** и **Gamer** отношение ассоциации: с одним игроком связан один сеанс игры. Это отношение будет реализовано с помощью включения ссылки на ассоциированный класс игральной кости в список атрибутов класса игрока.

➢ **Реализуйте поля и конструкторы класса**

- Добавьте в поля класса закрытые переменные – имя игрока и ссылку на класс игральной кости:

```
class Gamer
{
    string Name;
    IgralnayaKost seans;
 ...
```

- Добавьте конструктор с параметром, принимающим имя игрока и инициализирующий ссылку сеанса игры:

```
public Gamer(string name)
{
    Name = name;
    seans = new IgralnayaKost();
}
```

➢ **Реализуйте методы класса**

- Добавьте метод, который реализует бросок кости вызовом соответствующего метода класса **IgralnayaKost**:

```
public int SeansGame()
{
   return seans.random();
}
```

- Переопределите метод **ToString** для отображения объекта:

```
public override string ToString()
{
   return Name;
}
```

➢ **В итоге класс может быть таким:**

```
class Gamer
{
    string Name;
    IgralnayaKost seans;
```

```
        public Gamer(string name)
        {
            Name = name;
            seans = new IgralnayaKost();
        }

        public int SeansGame()
        {
            return seans.random();
        }

        public override string ToString()
        {
            return Name;
        }
    }
```

- **Протестируйте модель ассоциации**

  - В методе **Main** класса **Program** создайте игрока:
    ```
    Gamer g1 = new Gamer("Niko");
    ```
  - В цикле смоделируйте шесть бросков игральной кости и выведите информацию о количестве очков и имени игрока:
    ```
    for (int i = 1; i <= 6; i++)
    Console.WriteLine("Выпало количество очков {0} для игрока {1}", g1.SeansGame(), g1.ToString());
    ```
  - Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 7. Реализация прогрессии

В этом упражнении требуется:

- определить **абстрактный** класс **Progression**, описывающий прогрессии,
- в этом классе определить абстрактный метод **GetElement** с целочисленным параметром k, возвращающий элемент прогрессии,
- определить два производных класса **ArithmeticProgression** и **GeometricProgression**, описывающие арифметическую и геометрическую прогрессии.
- в каждом из классов необходимо определить конструктор, задающий параметры прогрессии и перегрузить унаследованный метод **GetElement**.

Для тестирования работы данного класса используйте класс **Program,** в котором пользователь вычисляет указанный им элемент прогрессий.

Лабораторная работа 8. Использование интерфейсов при реализации иерархии классов

Цель работы

Использование интерфейсов при реализации иерархии классов как важного элемента объектно-ориентированного программирования и приобретение навыков реализации интерфейсов.

Упражнение 1. Создание и реализация интерфейса

В этом упражнении вы создадите интерфейс, определяющий поведение классов, которые будут его реализовывать.

Предполагается, что в библиотечной системе, разработанной в прошлой работе, есть необходимость реализовать возможность оформления подписки на периодические издания. Включение этой функциональности в базовый класс **Item** не является правильным решением, так как к изданиям, оформляющим подписку, не относятся, например, книги. Поэтому необходимо создать интерфейс, объявляющий возможность оформления подписки и тогда классы, для которых предполагается данная функциональность, должны будут реализовывать этот интерфейс.

- **Выполните подготовительные операции**
  - Создайте папку Lab08 и скопируйте в нее решение MyClass, созданное в прошлом упражнении.

- **Создайте интерфейс IPubs с требуемой функциональностью**
  - Откройте проект MyClass.sln в папке *install folder*\Labs\Lab08\.
  - Добавьте в проект новый интерфейс с именем **IPubs**: **Projects** (Проект) → **Add class** (Добавить класс). В окне **Добавление нового элемента** выберите **Интерфейс** и укажите его имя **IPubs**.
  - В интерфейсе **IPubs** объявите его функциональные члены – метод для проверки оформлена ли подписка на издание **Subs** и свойство **IfSubs** для оформления подписки:

```
interface IPubs
{
    void Subs();
    bool IfSubs { get; set;}
}
```

- **Реализуйте интерфейс в классе Magazine**
  - Откройте класс **Magazine** и добавьте интерфейс в список наследования:

```
class Magazine : Item, IPubs
{
```

  - Реализуйте свойство и метод, объявленные в интерфейсе:

```
public bool IfSubs { get; set; }
```

```
    public void Subs()
    {
       Console.WriteLine("Подписка на журнал \"{0}\":
{1}." , title, IfSubs);

    }
```

- **Протестируйте новую функциональность**

  - В методе **Main** класса **Program** добавьте для уже имеющегося журнала **mag1** установку свойству **IfSubs** значения, устанавливающую подписку и вызовите метод **Subs** для отображения информации о подписке:

```
    Magazine mag1 = new Magazine("О природе", 5,"Земля и
мы", 2014, 1235, true);
    mag1.TakeItem();
    mag1.Show();
    mag1.IfSubs = true;
    mag1.Subs();
```

  - Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Упражнение 2. Использование стандартных интерфейсов

В библиотеке классов .Net определено множество стандартных интерфейсов, задающих желаемую функциональность объектов. В этом упражнении вы примените интерфейс **IComparable,** который задает метод сравнения объектов по принципу больше и меньше, что позволяет переопределить соответствующие операции в рамках класса, наследующего интерфейс **IComparable**.

Сравнение и дальнейшая сортировка будет реализована по полю **invNumber** - *Инвентарный номер*.

- **Реализуйте наследование интерфейса IComparable**

  - Добавьте в объявление абстрактного класса **Item** наследование интерфейса **IComparable:**

```
    abstract class Item : IComparable
    {
    . . .
```

Интерфейс **IComparable** определен в пространстве имен **System** и содержит единственный метод **CompareTo**, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра. Реализация данного метода должна возвращать: 0 – если текущий объект и параметр равны, отрицательное число, если текущий объект меньше параметра и положительное число, если текущий объект больше параметра.

- Добавьте в класс **Item** реализацию этого метода, причем сравнение реализуйте по полю **invNumber:**

```
int IComparable.CompareTo(object obj)
{
    Item it = (Item)obj;
    if (this.invNumber == it.invNumber) return 0;
  else if (this.invNumber > it.invNumber) return 1;
    else return -1;
}
```

➢ **Протестируйте использование новой функциональности**

- В методе **Main** класса **Program** создайте массив ссылок на абстрактный базовый класс **Item:**

```
Item[] itmas = new Item[4];
```

- Заполните массив созданными ранее книгами и журналом:

```
itmas[0] = b1;
itmas[1] = b2;
itmas[2] = b3;
itmas[3] = mag1;
```

- Отсортируйте массив с помощью статического метода **Sort** класса **Array**:

```
Array.Sort(itmas);
```

- Отобразите весь список книг и журналов, используя полиморфный вызов метода **Show**:

```
Console.WriteLine("\nСортировка по инвентарному
номеру");
foreach (Item x in itmas)
{
     x.Show();
}
```

- Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу. Информация о каждом элементе хранения должна выводиться согласно возрастанию инвентарных номеров.

Упражнение 3. Реализация прогрессии с помощью интерфейса

В этом упражнении замените абстрактный класс **Progression** из упражнения 7 предыдущей лабораторной работы на интерфейс **IProgression**, определяющий поведение классов **ArithmeticProgression** и **GeometricProgression**, описывающих арифметическую и геометрическую прогрессии. Внесите соответствующие изменения в текст и протестируйте работу программы.

Лабораторная работа 9. Применение делегатов и событий

Упражнение 1. Использование делегата при вызове метода

В этом упражнении вы реализуете в классе **Book** возможность вызова метода через делегат.

Делегат можно рассматривать как указатель на функцию, в качестве которой может выступать статический метод или метод экземпляра. При вызове делегата вызывается метод экземпляра по содержащейся в нем объектной ссылке. Создавая делегат, вы создаете объект, который может содержать ссылку на метод и этот метод можно вызвать посредством соответствующей ссылки. Таким образом, делегат может вызывать метод, на который он ссылается.

- **Выполните подготовительные операции**
  - Создайте папку Lab09 и скопируйте в нее решение MyClass из папки Lab07, созданное ранее.

- **Объявите делегат для класса Book**
  - Откройте проект MyClass.sln в папке *install folder*\Labs\Lab09\.
  - В класс Book добавьте объявление делегата

```
class Book : Item
{
 public delegate void ProcessBookDelegate(Book book);
```

  - Добавьте в проект новый класс с именем **Operation.** Он будет содержать методы с которыми будет связан делегат.
  - В класс **Operation** добавьте статический метод, который вызывает отображение информации о книге:

```
class Operation
{
   public static void PrintTitle(Book b)
   {
      b.Show();
   }
 }
```

  - Предполагается, что вызов метода будет для тех книг, которые были возвращены в срок. Контроль за возвращением удобно проводить с помощью свойства, добавьте в класс **Book** следующее автоматическое свойство:

```
public bool ReturnSrok { get; set; }
```

  - Для устранения неоднозначности удалите ранее объявленное поле **returnSrok** и созданный метод **ReturnSrok,** а также в методе **Return** замените проверку поля **returnSrok** на свойство **ReturnSrok.**

➢ **Создайте экземпляр делегата и свяжите его с методом**

- В классе **Book** добавьте метод, который будет находить возвращенные книги в срок и вызывать делегат для каждой книги:

```
    public void ProcessPaperbackBooks(ProcessBookDelegate
processBook)
    {
          if (ReturnSrok)
          processBook(this);
    }
```

- Постройте приложение и устраните ошибки.

➢ **Протестируйте вызов метода через делегат**

- В методе **Main** класса **Program** создайте два объекта – книга:

```
    Book b4 = new Book("Толстой Л.Н.", "Анна Каренина",
"Знание", 1204, 2014, 103, true);
    Book b5 = new Book("Неш Т", "Программирование для
профессионалов", "Вильямс", 1200, 2014, 108, true);
```

- Укажите, что первая книга возвращена в срок, а вторая нет:

```
    b4.ReturnSrok = true;
    b5.ReturnSrok = false;
```

- Вызовите статический метод **PrintTitle** посредством делегата для каждого объекта:

```
    System.Console.WriteLine("\nКниги возвращены в
срок:");
    b4.ProcessPaperbackBooks(Operation.PrintTitle);
    b5.ProcessPaperbackBooks(Operation.PrintTitle);
```

- Постройте и запустите приложение. Отобразится информация о первой из двух книг.

Упражнение 2. Работа с событиями

В этом упражнении вы в классе **Book** объявите событие “возвращение книги в срок”, и тогда объекты этого класса смогут уведомлять объекты других классов о данном событии. Событие – это автоматическое уведомление о выполнении некоторого действия.

➢ **Объявите событие и реализуйте условия его наступления**

- В классе **Book** объявите событие “возвращение книги в срок” на основе уже имеющегося делегата. В этом случае делегат будет событийным и использоваться для поддержки объявляемого события:

```
    class Book : Item
    {
      public delegate void ProcessBookDelegate(Book book);
      public static event ProcessBookDelegate RetSrok;
```

- Реализуйте возможность наступления события при установки свойству **ReturnSrok** значения **true**.

- Для этого удалите автоматическое свойство **ReturnSrok**, добавленное в прошлом упражнении и добавьте закрытое поле **returnSrok** и соответствующее ему свойство **ReturnSrok**:

```
private bool returnSrok = false;

public bool ReturnSrok
{
    get
    {
        return returnSrok;
    }
    set
    {
        returnSrok = value;
    }
}
```

- В set-аксессор добавьте код вызова события при наступлении требуемого условия:

```
set
    {
        returnSrok = value;
        if (ReturnSrok == true)
         RetSrok(this);
    }
```

➢ **Реализуйте требуемый формат отображения информации о книге**

- Предполагается, что при наступлении события "возвращение книги в срок" будет отображаться информация о книге, поэтому в классе **Book** переопределите метод **ToString**:

```
public override string ToString()
{
    string str = this.title + ", " + this.author + "
Инв. номер " + this.invNumber;
    return str;
}
```

➢ **Добавьте метод обработчик события – он будет вызываться при наступлении события**

- В класс **Operation** добавьте метод обработчик события – он будет вызываться при наступлении события

```
public static void MetodObrabotchik(Book b)
{
 Console.WriteLine("Книга {0} сдана в срок.",
b.ToString());
}
```

➢ **Протестируйте реакцию на событие**

- В методе **Main** класса **Program** после создания объектов **b4** и **b5** (обязательно перед вызовом свойства **ReturnSrok**) добавьте код,

реализующий подписку объектов класса **Book** на событие, так как событие было объявлено статическим, то подписка производится от имени класса:

```
    Book.RetSrok += new
Book.ProcessBookDelegate(Operation.MetodObrabotchik);
```

- Обратите внимание, что для книг **b4** и **b5** указано, что первая книга возвращена в срок, а вторая нет.
- Постройте и запустите приложение. Наступит событие “возвращение книги в срок” и отобразится информация о сданной книге в соответствии с переопределенным методом **ToString.**
- Укажите, что и книга b5 тоже сдана в срок:

```
    b5.ReturnSrok = true;
```

- Постройте и запустите приложение. Наступит событие “возвращение книги в срок” и отобразится информация обо всех сданных книгах.

Упражнение 3. Реализация события

В этом упражнении требуется в проекте **IgralnayaKost** лабораторная работа 7, упражнение 6) реализовать возникновение события «выпало максимальное количество очков» при броске игрального кубика.

Задание на самостоятельную работу. Иерархия классов учебного центра

В этом задании требуется реализовать иерархию классов учебного центра.

Создайте абстрактный класс **Person** с методами, позволяющим вывести на экран информацию о персоне, а также определить ее возраст (на момент текущей даты).

Создайте производные классы, моделирующие следующие сущности:

- Администратор (фамилия, дата рождения, лаборатория),
- Студент (фамилия, дата рождения, факультет, курс),
- Преподаватель (фамилия, дата рождения, факультет, должность, стаж),
- Менеджер (фамилия, дата рождения, факультет, должность)

со своими методами вывода информации на экран, и определения возраста.

Создайте интерфейс **IEmployee**, определяющий поведение классов – сотрудников учебного центра, которые будут его реализовывать. Поведение продумайте самостоятельно.

Создайте коллекцию персон, выведете полную информацию из коллекции на экран, а также организуйте поиск персон, чей возраст попадает в заданный диапазон.

Приложение

Таблица 1. Параметры форматирования C#

Параметр	**Значение**	
C или c	Денежный (currency) формат	Console.WriteLine(“Currency formatting – {0:C}”, 88.8); Currency formatting – $88.80
D или d	Десятичный формат. Позволяет задать общее количество знаков (при необходимости число дополняется слева нулями).	Console.WriteLine(“Integer formatting – {0:D5}”, 88); Integer formatting – 00088
E или e	Экспоненциальный формат	Console.WriteLine(“Exponental formatting – {0:E}”, 888.8); Exponental formatting – 8.888000E+002
F или f	Формат с фиксированной точностью. Позволяет задать количество знаков после запятой.	Console.WriteLine(“Fixed-point formatting – {0:F3}”, 888.8888); Fixed-point formatting – 888.889
G или g	Общий (general) формат. Применяется для вывода значений с фиксированной точностью или в экспоненциальном формате.	Console.WriteLine(“General formatting – {0:g}”, 888.8888); General formatting – 888.8888
N или n	Стандартное числовое форматирование с использованием разделителей (запятых) между разрядами.	Console.WriteLine(“Number formatting – {0:n}”, 8888888.8); Number formatting – 8,888,888.80
X или x	Шестнадцатеричный формат	Console.WriteLine(“Hexadecimal formatting – {0:X4}”, 88); Hexadecimal formatting – 0058

В общем виде синтаксис для форматирующей строки выглядит следующим образом: {N,M:FormatString}, где N – номер параметра, M – ширина поля и выравнивание, FormatString определяет формат выводимых данных.

Таблица 2. Флаги компиляции компилятора командной строки C#

Параметр командной строки	**Назначение**
/out:<file>	Определяет имя исполняемого файла (если не указано - производное от имени первого исходного файла)
/main:<тип>	Определяет класс, содержащий точку входа в программу (все остальные будут игнорироваться) (Краткая форма: /m)
/optimize[+|-]	Включает или отключает оптимизацию кода. (Краткая форма: /o).
/warn:<n>	Устанавливает уровень предупреждений компилятора (0-4) (Краткая форма: /w)
/warnaserror[+|-]	Рассматривает все предупреждения как ошибки
/target	Определяет тип сгенерированного приложения
/doc	Генерирует документацию в XML-файл
/debug[+|-]	Генерирует debug-информацию
/?, /help	Выводит информацию об опциях компилятора

Список литературы

1. Джеффри Рихтер CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C#. – СПб.: Питер, 2016. – 896 с.
2. Джон Скит C# для профессионалов. Тонкости программирования. – М.:Вильямс, 2014. – 408 с.
3. Фленов М. Библия C#/3-е издание. – СПб.: БХВ-Петербург, 2016. – 544 с.
4. Троелсен Э., Джепикс Ф. Язык программирования C# 6.0 и платформа .NET 4.6. – М.:Вильямс, 2016. – 1440 с.
5. Джозеф Албахари, Бен Албахари C# 6.0. Справочник. Полное описание языка– М.:Вильямс, 2016. – 1040 с.