МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Н.А. Осипов

Разработка приложений на C#

Учебное пособие

Санкт-Петербург

2012

УДК 004.655, 004.657, 004.62

Н.А. Осипов

Разработка приложений на C# - СПб: НИУ ИТМО, 2012. – 118 с.

В пособии излагаются методические указания к выполнению лабораторных работ по дисциплине «Технологии программирования».

Предназначено для студентов, обучающихся по всем профилям подготовки бакалавров направления: 210700 Инфокоммуникационные технологии и системы связи.

Рекомендовано к печати Ученым советом факультета Инфокоммуникационных технологий, протокол № 4 от 13 декабря 2011г.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики».

Оглавление

Введение

В результате изучения дисциплины «Технологии программирования» студенты познакомятся:

- с базовыми концепциями и терминологией объектно-ориентированного программирования;
- с основами семантики и синтаксиса языка программирования C#;
- с инструментальными средствами разработки программного обеспечения;

и приобретут навыки:

- разработки технических заданий на проектирование программного обеспечения;
- разработки алгоритма и реализации программного обеспечения на основе современных средств Microsoft Visual Studio;
- использования стандартных средств отладки программ.

По окончании обучения студенты смогут:

- понимать основные элементы .NET Framework и связь C# с элементами платформы .NET;
- работать в среде разработки Microsoft Visual Studio;
- создавать, отлаживать, компилировать и выполнять программы;
- создавать и использовать переменные;
- использовать выражения языка и применять обработку исключений;
- создавать методы;
- создавать, инициализировать и использовать массивы;
- знать базовые концепции и терминологию объектно-ориентированного программирования;
- создавать, инициализировать и разрушать объекты в программе;
- создавать классы и иерархии классов;
- определять операции и события в пользовательском классе;
- реализовывать свойства и индексаторы;
- использовать стандартные и пользовательские атрибуты.

Для выполнения упражнений используются готовые стартовые проекты. Исходные файлы для выполнения заданий находятся в сетевой папке \\atec\student\Work\C#\Console_App. Скопируйте их на локальную машину перед выполнением упражнений.

Лабораторная работа 1. Создание простой C#-программы

Цель работы

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

Упражнение 1. Создание простой программы

В этом упражнении Вы напишите программу на языке C#, используя среду разработки Visual Studio.NET. Программа будет спрашивать, как Вас зовут и затем здороваться с Вами по имени.

- **Создайте новое консольное C#-приложения**
  - 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\Lab02** 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.
  - Напишите код, считывающий введенное пользователем имя и присваивающий полученное значение строковой переменной *myName*.
  - Добавьте код, который будет выводить на экран строку “Hello *myName*”, где *myName* – имя, введенное пользователем.
  - 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 2. Компиляция и запуск C#-программы из командной строки

В этом упражнении Вы откомпилируете и запустите Вашу программу из командной строки.

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

- Запустите **Visual Studio .NET Command Prompt**. **(Start→All Programs→Visual Studio .NET→Visual Studio .NET Tools→Visual Studio .NET Command Prompt)**
- Перейдите в каталог *install folder*\Labs\Lab02\Greetings
- Откомпилируйте Вашу программу, используя следующую команду:

```
csc /out:Greet.exe Program1.cs
```

- Запустите программу, набрав в командной строке ее название:

```
Greet
```

- Закройте окно командной строки.

Упражнение 3. Использование отладчика Visual Studio .NET

В этом задании Вы приобретете навыки работы с интегрированным отладчиком Visual Studio .NET, изучите порядок отладки программы по шагам и окна для просмотра 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\Lab02\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.**
- 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 новую C#-программу**

- Запустите **Visual Studio .NET,** если она не запущена.
- 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\Lab02** 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.
- Напишите код, считывающий введенное пользователем число и присваивающий полученное значение переменной *temp* типа **string**.

- 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.
- Снова запустите программу на выполнение, нажав **Ctrl+F5.**
- Введите первое число **10** и нажмите **ENTER.**
- Введите второе число **0** и нажмите **ENTER.**
- В программе возникнет исключительная ситуация (деление на ноль).
- Для очистки окна диалога **Just-In-Time Debugging** выберите No.

➢ **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);
}
...
```

- 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);
   }
}
```

➢ **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.

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 это

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\Lab03\Starter\BankAccount.
- Переменуйте 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 типы Checking и Deposit.
- В методе **Main о**бъявите 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 1) 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\Lab03\Starter\StructType.
- Переменуйте 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:

Тип	Переменная
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.

Упражнение 3. Добавление возможности ввода/вывода

В этом упражнении Вы измените код, написанный в упражнении 2. Вместо использования счета номер 123, Вы будете запрашивать номер счета у пользователя, а потом использовать его при выводе информации о банковском счете на консоль.

➢ **Добавьте возможность ввода/вывода**

- Откройте (если он не открыт) проект StructType.sln из папки
install folder\Labs\Lab03\Starter\ StructType.
- Откройте файл Struct.cs и замените следующую строку:
```
goldAccount.accNo = 123; //remove this line and add code below
```
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.

Лабораторная работа 3. Использование выражений и исключений

Цель работы

Изучение и приобретение навыков использования управляющих конструкций для организации вычислений и механизма обработки исключительных ситуаций.

Упражнение 1. Преобразование дня года в дату типа месяц - день

В этом упражнении Вы напишите программу, которая считывает целое число, являющееся днем года (от 1 до 365) с экрана консоли и сохраняет его в целочисленной переменной. Далее программа преобразует это число в название и день месяца и выводит результат на консоль. Например, вводим числом 40, получаем результат “February 9”. (В данном упражнении не учитываются високосные годы.)

- **Считайте целое число, являющееся днем года, с экрана консоли**
  - Откройте проект WhatDay1.sln из папки *install folder*\Labs\Lab04\Starter\WhatDay1. В классе **WhatDay** имеется переменная, в которой в виде коллекции хранится количество дней каждого месяца.
  - В метод **WhatDay.Main** добавьте инструкцию **System.Console.Write,** которая запрашивает у пользователя день года от 1 до 365.
  - В методе **Main** объявите переменную *line* типа **string** и проинициализируйте ее значением, считанным с консоли с помощью метода **System.Console.ReadLine**.
  - В методе **Main** объявите переменную *dayNum* типа **int** и проинициализируйте ее целочисленным значением, возвращаемым методом **int.Parse**.

    Итоговый текст программы должен выглядеть следующим образом:

```
using System;
class WhatDay
{
   static void Main( )
   {
Console.Write("Please enter a day number between 1 and 365: ");
        string line = Console.ReadLine( );
        int dayNum = int.Parse(line);
   }
   ...
}
```

  - Сохраните проект.
  - Откомпилируйте программу WhatDay1.cs и исправьте ошибки, если это необходимо. Запустите программу.

- **Рассчитайте пару месяц-день для указанного дня года**
  - В методе **Main** объявите переменную *monthNum* типа **int** и проинициализируйте ее нулем.
  - Раскомментируйте 10 **if-**инструкций (одна для каждого месяца с января по октябрь) и добавьте еще две для ноября и декабря по аналогии.

Замечание: многострочные комментарии можно удалить, выделив необходимые строки и выбрав пункт меню **Edit→Advanced→Uncomment Selection**.

- В методе **Main** после последней инструкции **if** добавьте метку **End**.
- После метки **End** объявите переменную *monthName* типа **string** и не присваивайте ей начального значения.
- Раскомментируйте 10 **case-**ветвей оператора **switch** (одна для каждого месяца с января по октябрь), располагающихся после метки **End**, и добавьте еще две для ноября и декабря по аналогии. Также добавьте **default-**ветвь, в которой переменной *monthName* присваивается строковый литерал “not done yet”.
- После инструкции **switch** с помощью метода **WriteLine** выведите на консоль значения переменных *dayNum* и *monthName.*
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;

class WhatDay
{
   static void Main( )
   {
Console.Write("Please enter a day number between 1 and 365: ");
    string line = Console.ReadLine( );
    int dayNum = int.Parse(line);
    int monthNum = 0;
    if (dayNum <= 31) { // January
         goto End;
    } else {
         dayNum -= 31;
         monthNum++;
    }
    if (dayNum <= 28) { // February
         goto End;
    } else {
         dayNum -= 28;
         monthNum++;
    }
    if (dayNum <= 31) { // March
         goto End;
    } else {
         dayNum -= 31;
         monthNum++;
    }
    if (dayNum <= 30) { // April
         goto End;
        } else {
         dayNum -= 30;
         monthNum++;
    }
    if (dayNum <= 31) { // May
         goto End;
    } else {
         dayNum -= 31;
         monthNum++;
```

```
}
if (dayNum <= 30) { // June
      goto End;
     } else {
      dayNum -= 30;
      monthNum++;
}
if (dayNum <= 31) { // July
      goto End;
} else {
      dayNum -= 31;
      monthNum++;
}
if (dayNum <= 31) { // August
      goto End;
} else {
      dayNum -= 31;
      monthNum++;
}
if (dayNum <= 30) { // September
      goto End;
} else {
      dayNum -= 30;
      monthNum++;
}
if (dayNum <= 31) { // October
      goto End;
} else {
      dayNum -= 31;
      monthNum++;
}
     if (dayNum <= 30) { // November
      goto End;
} else {
      dayNum -= 30;
      monthNum++;
}
if (dayNum <= 31) { // December
      goto End;
} else {
      dayNum -= 31;
      monthNum++;
}
End:
   string monthName;
switch (monthNum) {
case O :
      monthName = "January"; break;
case 1 :
      monthName = "February"; break;
case 2 :
      monthName = "March"; break;
case 3 :
      monthName = "April"; break;
case 4 :
      monthName = "May"; break;
case 5 :
```

```
            monthName = "June"; break;
        case 6 :
            monthName = "July"; break;
        case 7 :
            monthName = "August"; break;
        case 8 :
            monthName = "September"; break;
        case 9 :
            monthName = "October"; break;
        case 10 :
            monthName = "November"; break;
        case 11 :
            monthName = "December"; break;
        default:
            monthName = "not done yet"; break;
        }

        Console.WriteLine("{0} {1}", dayNum, monthName);
      }
      ...
  }
```

- Сохраните проект.
- Откомпилируйте программу WhatDay1.cs и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что программа работает корректно, используя, например, данные из следующей таблицы:

День года	**Месяц и день**
32	February 1
60	March 1
91	April 1
186	July 5
304	October 31
309	November 5
327	November 23
359	December 25

➢ **Рассчитайте название месяца, используя перечисление**

- Замените инструкцию **switch**, с помощью которой рассчитывается название месяца более компактной конструкцией. Объявите перечисление **MonthName** и заполните его названиями месяцев с января по декабрь.
- Закомментируйте всю инструкцию **switch**.

  Замечание: можно создавать многострочные комментарии, выделив необходимые строки и выбрав пункт меню **Edit→Advanced→Comment Selection**.
- Вместо инструкции **switch** объявите переменную *temp* типа **MonthName**. Проинициализируйте ее целочисленным значением

переменной *monthNum***.** При этом необходимо будет использовать следующее выражение преобразования типов:

```
(MonthName)monthNum
```

- Замените инициализацию переменной *monthName* следующим выражением:

```
temp.ToString( )
```

- Итоговый текст программы должен выглядеть следующим образом:

```
using System;
enum MonthName
{
   January,
   February,
   March,
   April,
   May,
   June,
   July,
   August,
   September,
   October,
   November,
   December
}
class WhatDay
{
   static void Main( )
   {
Console.Write("Please enter a day number between 1 and 365: ");
         string line = Console.ReadLine( );
         int dayNum = int.Parse(line);
         int monthNum = 0;
         // 12 if statements, as above
         End:
         MonthName temp = (MonthName)monthNum;
         string monthName = temp.ToString( );

         Console.WriteLine("{0} {1}", dayNum, monthName);
   }
   ...
}
```

- Сохраните проект
- Откомпилируйте программу WhatDay1.cs и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что программа все еще работает корректно, используя данные из предыдущей таблицы.

➢ **Замените 12 if-инструкций одним циклом foreach**

- Замените 12 выражений, рассчитывающих пары день-месяц, одним циклом **foreach**. Закомментируйте все 12 **if-**инструкций.
- Напишите цикл **foreach**, который опрашивает элементы коллекции **DaysInMonths.** Для этого используйте следующее выражение:

```
foreach (int daysInMonth in DaysInMonths) ...
```

- В тело цикла **foreach** добавьте блок операторов. Содержимое этого блока будет очень похоже на отдельные закомментированные **if-**инструкции, только вместо различных числовых литералов будет использоваться переменная *daysInMonth*.
- Закомментируйте метку **End,** которая располагается над закомментированной инструкцией **switch.** В цикле **foreach** замените выражение **goto** на **break.**
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;
enum MonthName { ... }
class WhatDay
{
   static void Main( )
   {
Console.Write("Please enter a day number between 1 and 365: ");
            string line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            int monthNum = 0;

            foreach (int daysInMonth in DaysInMonths) {
             if (dayNum <= daysInMonth)
             {
                   break;
             }
             else
             {
                   dayNum -= daysInMonth;
                   monthNum++;
             }
            }
            MonthName temp = (MonthName)monthNum;
            string monthName = temp.ToString( );

            Console.WriteLine("{0} {1}", dayNum, monthName);
   }
   …
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay1.cs и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что программа все еще работает корректно, используя данные из предыдущей таблицы.
- Запустите программу. Введите значение меньше 1 или больше 365 и посмотрите, что произойдет.

Упражнение 2. Проверка вводимого пользователем значения дня года

В этом упражнении Вы расширите возможности программы, созданной в упражнении 1. Программа будет проверять день года, вводимого пользователем. Если он будет меньше 1 или больше 365, то

будет выбрасываться исключение **InvalidArgument** (“Day out of range”). Программа будет перехватывать это исключение в блоке **catch** и выводить на консоль сообщение об ошибке.

- **Проверьте вводимое пользователем значение дня года**
  - Откройте проект WhatDay2.sln из папки *install folder*\Labs\Lab04\Starter\WhatDay2.
  - Поместите все содержимое метода **WhatDay.Main** внутрь блока **try**.
  - После блока **try** добавьте блок **catch,** который перехватывает исключения типа **System.Exception** с именем **caught**. Внутри блока **catch** добавьте метод **WriteLine,** выводящий на консоль информацию о перехваченном исключении.
  - После объявления переменной *dayNum* добавьте инструкцию **if**, которая будет выбрасывать новое исключение **System.ArgumentOutOfRangeException,** если значение *dayNum* меньше 1 или больше 365. Для создания объекта исключения используйте строковую константу **“**Day out of range**”.**
  - Итоговый текст программы должен выглядеть следующим образом:

```
using System;

enum MonthName { ... }

class WhatDay
{
   static void Main( )
   {
         try      {
              Console.Write("Please enter a day number
          between 1 and 365: ");
              string line = Console.ReadLine( );
              int dayNum = int.Parse(line);

              if (dayNum < 1 || dayNum > 365) {
              throw new ArgumentOutOfRangeException("Day
          out of range");
                   }

              int monthNum = 0;

              foreach (int daysInMonth in DaysInMonths) {
               if (dayNum <= daysInMonth) {
                    break;
               } else {
                    dayNum -= daysInMonth;
                    monthNum++;
               }
              }
              MonthName temp = (MonthName)monthNum;
              string monthName = temp.ToString( );

              Console.WriteLine("{0}      {1}",      dayNum,
          monthName);
```

```
            }
            catch (Exception caught) {
             Console.WriteLine(caught);
            }
        }
        ...
    }
```

- Сохраните проект.
- Откомпилируйте программу WhatDay2.cs и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что программа работает корректно, используя данные из таблицы, приведенной в Лабораторной работе 3.1 (упражнение 1).
- Запустите программу, введите значения дня года меньше 1 и больше 365. Убедитесь в том, что некорректный ввод данных обрабатывается и исключение выбрасывается, перехватывается и выводится на экран.

Упражнение 3. Учет високосных годов

В этом упражнении Вы расширите возможности программы, разработанной в упражнении 2. Конечный вариант программы будет запрашивать у пользователя не только день года, но и сам год. Программа будет определять, является ли год високосным. Если да, то будет проверяться, попадает ли значение дня года в диапазон от 1 до 366. Если год не является високосным, то проверяется попадание значения дня года в диапазон от 1 до 365.

➢ **Считайте значение года с экрана консоли**

- Откройте проект WhatDay3.sln из папки *install folder*\Labs\Lab04\Starter\WhatDay3.
- В начале метода **System.Main** добавьте инструкцию **System.Console.Write**, запрашивающую у пользователя значение года.
- Замените строку

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

  на

  ```
  line = Console.ReadLine( );
  ```

- В методе **Main** объявите переменную *line* типа **string** и проинициализируйте ее значением, считанным с консоли с помощью метода **System.Console.ReadLine**.
- В методе **Main** объявите переменную *yearNum* типа **int** и проинициализируйте ее целочисленным значением, возвращаемым методом **int.Parse**.
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;

enum MonthName { ... }
```

```
class WhatDay
{
      static void Main( )
      {
            try {
                  Console.Write("Please enter the year: ");
                  string line = Console.ReadLine( );
                  int yearNum = int.Parse(line);
                  Console.Write("Please enter a day number
             between 1 and 365: ");
                  line = Console.ReadLine( );
                  int dayNum = int.Parse(line);

                  // As before....
            }
            catch (Exception caught) {
                  Console.WriteLine(caught);
            }
      }
      ...
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay3.cs и исправьте ошибки, если это необходимо.

➢ **Определите, является ли введенный пользователем год високосным**

- Сразу после объявления переменной *yearNum* объявите переменную *isLeapYear* типа **bool.** Присвойте ей значение логического выражения, определяющего является ли *yearNum* високосным годом. Год является високосным, если одновременно выполняются два условия:
  - Число кратно 4.
  - Число *не* делится на 100 *или* делится на 400.
- Сразу после объявления переменной *isLeapYear* добавьте инструкцию **if**, которая будет выводить на консоль строку “IS a leap year” или “is NOT a leap year”, в зависимости от значения переменной *isLeapYear*. Эта **if-**инструкция поможет оценить корректность, написанного Вами логического выражения, определяющего, является ли год високосным.
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;

enum MonthName { ... }

class WhatDay
{
      static void Main( )
      {
            try
```

```
            {
                  Console.Write("Please enter the year: ");
                  string line = Console.ReadLine( );
                  int yearNum = int.Parse(line);

                  bool isLeapYear = (yearNum % 4 == 0)
                   && (yearNum % 100 != 0
                         || yearNum % 400 == 0);

                  if (isLeapYear)
                  {
                   Console.WriteLine(" IS a leap year");
                  } else
                  {
                   Console.WriteLine(" is NOT a leap year");
                  }

                  Console.Write("Please enter a day number
               between 1 and 365: ");
                  line = Console.ReadLine( );
                  int dayNum = int.Parse(line);

                  // As before...
            }
            catch (Exception caught)
            {
             Console.WriteLine(caught);
            }
      }
      ...
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay3.cs и исправьте ошибки, если это необходимо. Убедитесь в том, что логическое выражение, определяющее, является ли год високосным, работает корректно, используя данные из следующей таблицы.

Високосный год	***Не високосный год***
1996	1999
2000	1900
2004	2001

- Закомментируйте **if-**инструкцию, созданную в этом упражнении.

➢ **Проверьте, что введенное пользователем значение дня года попадает в необходимый диапазон (от 1 до 365 или от 1 до 366)**

- Сразу после объявления переменной *isLeapYear*, объявите переменную *maxDayNum* типа **int**. Присвойте переменной значение 366 или 365, в зависимости от значения переменной *isLeapYear* (**true** или **false**, соответственно).

- Измените инструкцию **WriteLine**, запрашивающую у пользователя день года. Она должна выводить диапазон от 1 до 366 для високосного года и от 1 до 365 для не високосного года.
- Измените **if-**инструкцию, проверяющую значение *dayNum*, так, чтобы она использовала вместо 365 переменную *maxDayNum*.
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;
enum MonthName { ... }
class WhatDay
{
      static void Main( )
      {
            try
            {
             Console.Write("Please enter the year: ");
                  string line = Console.ReadLine( );
                  int yearNum = int.Parse(line);
                  bool isLeapYear = (yearNum % 4 == 0)
                   && (yearNum % 100 != 0
                         || yearNum % 400 == 0);
                  int maxDayNum = isLeapYear ? 366 : 365;

                  Console.Write("Please enter a day number
             between 1 and {0}: ", maxDayNum);
                  line = Console.ReadLine( );
                  int dayNum = int.Parse(line);

                  if (dayNum < 1 || dayNum > maxDayNum) {
                   throw                                  new
             ArgumentOutOfRangeException("Day    out    of
             range");
                  }
                  // As before....
            }
            catch (Exception caught)
            {
             Console.WriteLine(caught);
            }
      }
      ...
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay3.cs и исправьте ошибки, если это необходимо. Убедитесь в том, что задание выполнено корректно.

➢ **Рассчитайте пару месяц-день для високосных годов**

- После **if**-инструкции, проверяющей введенное пользователем значение дня года и объявления целочисленной переменной *monthNum*, добавьте инструкцию **if-else**. Логическим условием в этой инструкции будет переменная *isLeapYear*.

- Переместите цикл **foreach** внутрь обеих ветвей инструкции **if-else** (для **true** и **false**). После этого шага ваш код должен выглядеть следующим образом:

```
if (isLeapYear)
{
   foreach (int daysInMonth in DaysInMonths) {
   ...
   }
} else
{
   foreach (int daysInMonth in DaysInMonths) {
   ...
   }
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay3.cs и исправьте ошибки, если это необходимо. Запустите программу и убедитесь в том, что дни года для не високосных годов обрабатываются корректно.
- На следующем шаге вы воспользуетесь заранее созданной коллекцией **DaysInLeapMonths.** Эта коллекция состоит из целочисленных значений и похожа на коллекцию **DaysInMonths**. Единственное исключение – второе значение (количество дней в феврале) – 29, а не 28.
- В первой ветви инструкции **if-else** (условие **true**) вместо коллекции **DaysInMonths** используйте коллекцию **DaysInLeapMonths.**
- Итоговый текст программы должен выглядеть следующим образом:

```
using System;

enum MonthName { ... }

class WhatDay
{
      static void Main( )
      {
            try {
                  Console.Write("Please enter the year: ");
                  string line = Console.ReadLine( );
                  int yearNum = int.Parse(line);

                  bool isLeapYear = (yearNum % 4 == 0)
                   && (yearNum % 100 != 0
                         || yearNum % 400 == 0);

                  int maxDayNum = isLeapYear ? 366 : 365;

                  Console.Write("Please enter a day number
               between 1 and {0}: ", maxDayNum);
                  line = Console.ReadLine( );
                  int dayNum = int.Parse(line);

                  if (dayNum < 1 || dayNum > maxDayNum) {
```

```
                throw new
            ArgumentOutOfRangeException("Day out of
            range");
                }

                int monthNum = 0;

                if (isLeapYear) {
                 foreach (int daysInMonth in
            DaysInLeapMonths) {
                      if (dayNum <= daysInMonth) {
                           break;
                      } else {
                           dayNum -= daysInMonth;
                           monthNum++;
                      }
                 }
                } else {
                 foreach (int daysInMonth in DaysInMonths)
            {
                      if (dayNum <= daysInMonth) {
                           break;
                      } else {
                           dayNum -= daysInMonth;
                           monthNum++;
                      }
                 }
                }

                MonthName temp = (MonthName)monthNum;
                string monthName = temp.ToString( );
                Console.WriteLine("{0} {1}", dayNum,
            monthName);
           }
           catch (Exception caught) {
            Console.WriteLine(caught);
           }
      }
      ...
}
```

- Сохраните проект.
- Откомпилируйте программу WhatDay3.cs и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что программа работает корректно, используя данные из следующей таблицы:

Год	**День года**	**Месяц и день**
1999	32	February 1
2000	32	February 1
1999	60	March 1
2000	60	February 29
1999	91	April 1
2000	91	March 31
1999	186	July 5

Год	День года	Месяц и день
2000	186	July 4
1999	304	October 31
2000	304	October 30
1999	309	November 5
2000	309	November 4
1999	327	November 23
2000	327	November 22

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 создадите еще один класс (класс **Test**), в котором у пользователя будут запрашиваться два числа, далее будет вызываться метод **Utils.Greater**, после чего на экран консоли будет выводиться результат.

- **Создайте метод Greater**
  - Откройте проект Utils.sln из папки *install folder*\Labs\Lab05\Starter\Utility.

    В нем содержится пространство имен **Utils**, в котором создан класс **Utils**. Вам необходимо создать для этого класса метод **Greater**.
  - Создайте метод **Greater** следующим образом:
    - Откройте класс **Utils.**
    - В класс **Utils** добавьте **public static** метод **Greater**.
    - Этот метод будет использовать два передаваемых по значению параметра *a* и *b* типа **int** и возвращать значение типа **int**, являющееся большим из двух передаваемых значений.
  - Текст класса **Utils** должен выглядеть следующим образом:

```
namespace Utils
{
      using System;

      class Utils
      {
            //
            // Return the greater of two integer values
            //
```

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

- **Протестируйте метод Greater**
  - Откройте класс **Test**.
  - Внутри метода **Main** напишите следующий код:
    - Объявите две целочисленных переменных *x* и *y*.
    - Добавьте код для считывания введенных пользователем чисел, и сохраните их в переменных *x* и *y*. (Используйте методы **Console.ReadLine** и **int.Parse**)**.**
    - Объявите еще одну целочисленную переменную и назовите ее *greater*.
    - Протестируйте метод **Greater,** вызвав его на исполнение и присвоив возвращенное им значение переменной *greater*.
  - Напишите код, выводящий на консоль большее из двух чисел, используя метод **Console.WriteLine.**
  - Текст класса **Test** должен выглядеть следующим образом:

```
namespace Utils
{
    using System;

    /// <summary>
    /// This the test harness
    /// </summary>

    public class Test
    {
        public static void Main( )
        {
            int x; // Input value 1
            int y; // Input value 2
            int greater; // Result from Greater()

            // Get input numbers
            Console.WriteLine("Enter first number:");
            x = int.Parse(Console.ReadLine( ));
            Console.WriteLine("Enter second number:");
            y = int.Parse(Console.ReadLine( ));

            // Test the Greater( ) method
            greater = Utils.Greater(x,y);
            Console.WriteLine("The greater value is "+
             greater);
        }
    }
}
```

- Сохраните проект.
- Откомпилируйте 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**

- Откройте проект Utils.sln из папки *install folder*\Labs\Lab05\Starter\Utility, если он не открыт.
- Добавьте в класс **Utils** метод **Swap** следующим образом:
  - Добавьте **public static void** метод **Swap**.
  - Этот метод будет использовать два передаваемых по ссылке параметра *a* и *b* типа **int.**
  - 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.
- Текст класса **Utils** должен выглядеть следующим образом:

```
namespace Utils
{
      using System;

      public class Utils
      {
            // As before....

            //
            // Exchange two integers, passed by reference
            //

            public static void Swap(ref int a, ref int b)
            {
                  int temp = a;
                  a = b;
                  b = temp;
            }
      }
}
```

➢ **Протестируйте метод Swap**

- Отредактируйте метод **Main** класса **Test**, выполнив следующее:
  - Считайте значение для переменных *x* и *y*.
  - Вызовите метод **Swap**, передав эти значения в качестве параметров.

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

Текст класса **Test** должен выглядеть следующим образом:

```
namespace Utils
{
      using System;

      public class Test
      {
            public static void Main( )
            {
                  //As before…

                  // Test the Swap method
                  Console.WriteLine("Before swap: " + x +
                 "," + y);
                  Utils.Swap(ref x,ref y);
                  Console.WriteLine("After swap: " + x + ","
                 + y);
             }
      }
}
```

- Сохраните проект.
- Откомпилируйте 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 целое значение и рассчитывающий его факториал следующим образом:

- Factorial(0)=1
- Factorial(1)=1
- Factorial(2)=1*2=2
- Factorial(3)=1*2*3=6
- Factorial(4)=1*2*3*4=24

➢ **Создайте метод Factorial**

- Откройте проект Utils.sln из папки *install folder*\Labs\Lab05\Starter\Utility, если он не открыт.
- Добавьте в класс **Utils** метод **Factorial** следующим образом:
  - Добавьте **public static** метод **Factorial**.
  - Этот метод 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 (может произойти переполнение и выброс исключения).

- Внутри метода напишите код расчета факториала для передаваемого на вход значения.

  Проще всего рассчитать факториал, используя цикл следующим образом:

  - В методе **Factorial** создайте переменную *k* типа **int.** Она будет использоваться в цикле в качестве счетчика.
  - Создайте еще одну переменную типа **int,** назовите ее *f* и задайте ей начальное значение 1. Эта переменная будет использоваться внутри цикла.
  - Создайте цикл **for**. Начальное значение *k*=2, итерации продолжаются до тех пор, пока не будет достигнуто значение параметра *n*. На каждом шаге увеличивайте значение *k* на единицу.
  - В теле цикла умножайте *f* на *k* и сохраняйте результат в *f*.
  - Значение факториала растет достаточно быстро, поэтому производите проверку на арифметическое переполнение в блоке checked и, при необходимости перехватывайте исключения.
  - Итоговое значение переменной *f* присвойте возвращаемому параметру *answer*.
  - Если метод отработал успешно, он возвращает значение **true**, если произошло арифметическое переполнение (выброс исключения), то возвращается значение **false**.

  Текст класса **Utils** должен выглядеть следующим образом:

```
namespace Utils
{
      using System;

      public class Utils
      {
            //As before…
            // Calculate factorial
            // and return the result as an out parameter

       public static bool Factorial(int n, out int answer)
            {
                  int k; // Loop counter
                  int f; // Working value
             bool ok=true; // True if okay, false if not

                  // Check the input value

                  if (n<0)
                   ok = false;

// Calculate the factorial value as the
```

```
// product of all of the numbers from 2 to n

                    try
                    {
                     checked
                     {
                           f = 1;
                           for (k=2; k<=n; ++k)
                           {
                                 f = f * k;
                           }
                     }
                    }
                    catch(Exception)
                    {
                // If something goes wrong in the calculation,
                // catch it here. All exceptions
                // are handled the same way: set the result
                // to zero and return false.

                     f = 0;
                     ok = false;
                    }

                    // Assign result value
                    answer = f;
                    // Return to caller
                    return ok;
          }
        }
    }
```

- **Протестируйте метод Factorial**
  - Отредактируйте класс **Test**, выполнив следующее:
    - Объявите переменную *ok* типа **bool** для хранения возвращаемого методом значения (**true** или **false**).
    - Объявите переменную *f* типа **int** для хранения факториала числа, рассчитанного в методе.
    - Запросите у пользователя целое число. Сохраните введенное значение в переменной *x* типа **int**.
    - Вызовите метод **Factorial,** передав *x* и *f* в качестве параметров. Возвращаемое методом значение присвойте переменной *ok*.
    - Если *ok* принимает значение **true**, выведите на консоль значение *x* и *f*, в противном случае выведите на экран сообщение об ошибке.

  Текст класса **Test** должен выглядеть следующим образом:

```
namespace Utils
{
     public class Test
     {

          static void Main( )
          {
```

```
                    int f; // Factorial result
                    bool ok; // Factorial success or failure

                    //As before…

                    // Get input for factorial

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

                    // Test the factorial function
                    ok = Utils.Factorial(x, out f);
                    // Output factorial results
                    if (ok)
Console.WriteLine("Factorial(" + x + ") = " +f);
                    else
Console.WriteLine("Cannot compute this factorial");
                }
            }
        }
```

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

Лабораторная работа 5. Создание и использование массивов

Цель работы

Изучение массивов и приобретение навыков работы с ними.

Упражнение 1. Работа с массивами размерных типов.

В этом упражнении Вы напишите программу, в которой в метод **Main** в качестве аргумента будет передаваться имя текстового файла. Содержимое текстового файла будет считываться в массив символов, а дальше будут производиться итерации по всему массиву для подсчета количества гласных и согласных. В итоге, на консоль будет выводиться информация об общем количестве символов, гласных, согласных и строк.

- **Передайте в метод Main в качестве параметра имя текстового файла**
  - Откройте проект FileDetails.sln из папки *install folder*\Labs\Lab06\Starter\FileDetails.
  - В качестве параметра метода **Main** класса **FileDetails** добавьте массив строк **args**. В этом массиве будут содержаться все параметры командной строки, передаваемые при запуске программы. В этом задании в качестве аргумента командной строки в метод **Main** будет передаваться имя текстового файла.
  - В метод **Main** добавьте инструкцию для вывода на консоль длины массива **args**. Это позволит убедиться, что длина массива **args** равна

нулю, если во время запуска программы в метод **Main** не передаются никакие аргументы.

- В метод **Main** добавьте цикл **foreach**, в котором выводятся на консоль все строки массива **args.** Это поможет убедиться в том, что метод **Main** получает аргументы командной строки на этапе выполнения.

  Итоговый код метода **Main** должен выглядеть следующим образом:

```
static void Main(string[ ] args)
{
      Console.WriteLine(args.Length);
      foreach (string arg in args) {
       Console.WriteLine(arg);
      }
}
```

- Откомпилируйте программу FileDetails.cs и исправьте ошибки, если это необходимо. Запустите программу из командной строки, не передавая никаких аргументов командной строки. Убедитесь в том, что длина массива **args** равна нулю.

  Подсказка: для запуска программы из командной строки, откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab06\Starter\FileDetails\bin\Debug. Исполняемый файл находится в этой папке.
- Запустите программу из командной строки, передав в качестве аргумента командной строки имя файла *install folder*\Labs\Lab06\Starter\FileDetails\FileDetails.cs. Убедитесь в том, что на этапе выполнения имя файла передается в метод **Main.**
- Протестируйте программу, передавая различные аргументы, и убедитесь в том, что все они корректно выводятся на консоль. Закомментируйте инструкцию, выводящую информацию на консоль.
- В методе **Main** объявите переменную *fileName* типа **string** и присвойте ей значение `args[0]`.

➢ **Считайте содержимое текстового файла в массив**

- Снимите комментарии с кода, отвечающего за объявление и инициализацию объектов типа FileStream и StreamReader.
- Определите длину текстового файла.

  Подсказка: Чтобы найти соответствующее свойство класса Stream, читайте раздел “Stream class” в .NET Framework SDK Help documents.
- В методе **Main** объявите переменную символьного массива *contents*. Проинициализируйте ее новым экземпляром массива, длина которого равна только что определенной вами длине текстового файла.

- В метод **Main** добавьте цикл **for,** в теле которого будет считываться один символ из *reader* и добавляться в *contents*.

  Подсказка: воспользуйтесь методом **Read**, который не использует параметров и возвращает значение типа **int**. Перед тем как сохранять результат в массиве приведите его к типу **char**.
- В метод **Main** добавьте цикл **foreach**, в котором посимвольно будет выводиться на консоль все содержимое символьного массива. Это поможет убедиться в том, что содержимое текстового файла было успешно считано в массив *contents*.

  Итоговый код метода **Main** должен выглядеть следующим образом:

```
static void Main(string[ ] args)
{
     string fileName = args[0];
     FileStream   stream   =   new   FileStream(fileName,
 FileMode.Open);
     StreamReader reader = new StreamReader(stream);
     int size = (int)stream.Length;
     char[ ] contents = new char[size];
     for (int i = 0; i < size; i++) {
      contents[i] = (char)reader.Read( );
     }
     foreach(char ch in contents) {
      Console.Write(ch);
     }
}
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу из командной строки, передав в качестве аргумента командной строки имя файла *install folder*\Labs\Lab06\Starter\FileDetails\FileDetails.cs. Убедитесь в том, что содержимое файла корректно отображается на консоли.
- Закомментируйте цикл **foreach**.
- Закройте объект **Reader**, вызвав подходящий метод для **StreamReader.**

➢ **Систематизируйте и резюмируйте информацию о содержимом файла**

- В классе **FileDetails** объявите новый статический метод **Summarize.** Этот метод не будет возвращать значений и будет принимать в качестве параметра массив символов. В метод **Main** добавьте инструкцию вызова метода **Summarize**, передав *contents* в качестве аргумента.
- В метод **Summarize** добавьте цикл **foreach**, в котором будет просматриваться каждый символ массива, передаваемого в качестве аргумента. Подсчитайте количество гласных, согласных и символов перевода строки, сохраняя результаты в отдельных переменных.

Подсказка: Чтобы определить, является ли символ гласной, создайте строку из всех возможных гласных и воспользуйтесь для нее методом **IndexOf**, который позволит определить, существует ли в данной строке указанный символ:

```
if ("AEIOUaeiou".IndexOf(myCharacter) != -1) {
 // myCharacter is a vowel
} else {
 // myCharacter is not a vowel
}
```

- Напишите четыре инструкции для вывода на консоль следующей информации:
  - Общее количество символов в файле.
  - Общее количество гласных в файле.
  - Общее количество согласных в файле.
  - Общее количество строк в файле.

  Итоговый код метода **Summarize** должен выглядеть следующим образом:

```
static void Summarize(char[ ] contents)
{
      int vowels = 0, consonants = 0, lines = 0;
      foreach (char current in contents) {
            if (Char.IsLetter(current)) {
                  if ("AEIOUaeiou".IndexOf(current) != -1) {
                   vowels++;
                  } else {
                   consonants++;
                  }
            }
            else if (current == '\n') {
             lines++;
            }
      }
      Console.WriteLine("Total no of characters: {0}",
 contents.Length);
      Console.WriteLine("Total no of vowels : {0}",
 vowels);
      Console.WriteLine("Total no of consonants: {0}",
 consonants);
      Console.WriteLine("Total no of lines : {0}", lines);
}
```

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу из командной строки для файла с решением (хранящегося в папке **Solution**): *install folder*\Labs\Lab06\Solution\FileDetails\FileDetails.cs. Правильные результаты:
  - 1353 символа
  - 247 гласных
  - 394 согласных
  - 41 строка

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.

Формула расчета произведения матриц A и B:

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

- **Перемножьте две матрицы**
  - Откройте проект MatrixMultiply.sln из папки *install folder*\Labs\Lab06\Starter\MatrixMultiply.
  - В методе **Main** класса **MatrixMultiply** объявите массив целых чисел **a** размером 2x2. Позднее программа будет заполнять его числами, считанными с консоли. Но пока заполните его значениями из таблицы, приведенной ниже. Это позволит проверить, что перемножение матриц реализовано корректно.

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

  - Добавьте в метод **Main** объявление еще одного массива целых чисел размером 2x2, назовите его **b**. Позднее программа будет заполнять его числами, считанными с консоли. Но пока заполните его значениями из таблицы, приведенной ниже.

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

  - 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**
  - В классе **MatrixMultiply** объявите новый статический метод **Output**. Этот метод не возвращает значений и принимает в качестве параметра массив целых чисел второго ранга с именем **result**.
  - Вырежьте из метода **Main** четыре инструкции, выводящие на консоль значения массива **result,** и вставьте их в метод **Output.**
  - В метод **Main** добавьте вызов метода **Output,** передав в качестве аргумента массив **result.**

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

```
static void Output(int[,] result)
{
      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 те же четыре значения:

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

  - Измените метод **Output,** используя вместо четырех инструкций **Console.WriteLine** два вложенных цикла **for**. Для проверки верхних границ обоих массивов используйте число 2.

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

```
static void Output(int[,] result)
{
      for (int r = 0; r < 2; r++) {
       for (int c = 0; c < 2; 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:

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

- Снова измените метод **Output,** сделав его более общим. Замените число 2, используемое для проверки верхних границ каждого из массивов на вызов метода **GetLength.**

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

```
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:

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

➢ **Создайте метод, рассчитывающий и возвращающий значения массива result**

- В классе **MatrixMultiply** объявите новый статический метод **Multiply**. Этот метод будет возвращать массив целых чисел второго ранга и принимать в качестве параметров два массива целых чисел второго ранга с именами **a** и **b**.
- Скопируйте объявление и инициализацию массива **result** из метода **Main** в метод **Multiply**.
- В метод **Multiply** добавьте инструкцию **return**, возвращающую массив **result**.
- В методе **Main** замените инициализацию массива **return** на вызов метода **Multiply,** передав в качестве аргументов **a** и **b**.

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

```
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:

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

➢ **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 число 2. 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]
```

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

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

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

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

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

- В методе **Main** удалите инструкции инициализации массива **a**.
- В методе **Main** запросите у пользователя и считайте с консоли четыре целых значения для массива **a**. Эти инструкции должны быть

помещены выше вызова метода **Multiply**. Инструкции для считывания одного значения с консоли:

```
string s = Console.ReadLine( );
a[0,0] = int.Parse(s);
```

- 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:

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

- В классе **MatrixMultiply** объявите новый статический метод **Input**. Это метод не будет возвращать значений и будет принимать в качестве параметра массив целых чисел второго ранга с именем **dst**.
- Из метода **Main** вырежьте инструкции для считывания значений массива **a** и вставьте их в метод **Input**. В метод **Main** добавьте вызов метода **Input**, передавая в качестве аргумента массив **a**. Эта инструкция должна быть помещена выше вызова метода **Multiply**.
- 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:

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

- В методе **Input** для запроса значений массива **a** используйте два вложенных цикла **for**. Для проверки верхней границы массива используйте число 2. Во внутреннем цикле **for** для запроса у пользователя значений используйте инструкцию **Write**.

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

```
static void Input(int[,] dst)
{
      for (int r = 0; r < 2; r++) {
       for (int c = 0; c < 2; c++) {
       Console.Write("Enter value for [{0},{1}] : ", r, c);
            string s = Console.ReadLine( );
            dst[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:

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

- **Обеспечьте возможность считывания с консоли значений второй матрицы**
  - В методе **Main** удалите инструкции инициализации массива **b**.
  - В метод **Main** для считывания значений массива **b** добавьте вызов метода **Input**, передав в качестве аргумента массив **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:

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

  - Попробуйте вводить другие данные. Сравните свои результаты с результатами других студентов при вводе одинаковых данных.

Лабораторная работа 6. Создание и использование классов

Цель работы

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

Упражнение 1. Создание и использование класса

В этом упражнении Вы преобразуете структуру для описания банковского счета, созданную в предыдущих лабораторных работах, в класс.

Члены класса, описывающие данные, вы объявите с модификатором доступа **private**, для доступа к данным вы создадите нестатические **public** методы. Далее вы протестируете работу этого класса, создав объект банковского счета и заполнив его данными, введенными пользователем (номер счета и баланс счета). В итоге информация о данных счета должна выводиться на экран.

- **Преобразуйте структуру BankAccount в класс**
  - Откройте проект CreateAccount.sln из папки *install folder*\Labs\Lab07\Starter\CreateAccount.
  - Изучите код в файле BankAccount.cs. Обратите внимание на то, что тип *BankAccount* определен как структура.
  - Откомпилируйте и запустите программу. Вас попросят ввести номер и начальный баланс счета.
  - Преобразуйте структуру BankAccount, определенную в файле BankAccount.cs, в класс.

- Откомпилируйте программу. Вы получите сообщение об ошибке. Откройте файл CreateAccount.cs и изучите класс **CreateAccount**. Этот класс выглядит следующим образом:

```
class CreateAccount
{
      ...
      static BankAccount NewBankAccount( )
      {
            BankAccount created;
            ...
            created.accNo = number; // Error here
            ...
      }
      ...
}
```

- Инициализация *created.accNo* компилировалась без ошибки, когда **BankAccount** был определен как структура. Теперь это класс и компиляция не проходит. Это связано с тем, что когда **BankAccount** был структурой, при объявлении переменной *created*, создавалась размерная переменная типа **BankAccount** (в стеке). Теперь, когда **BankAccount** стал классом, при объявлении переменной *created* создается не размерная переменная **BankAccount**, а ссылка **BankAccount**, которая должна указывать на объект типа **BankAccount.**
- Измените объявление переменной *created* таким образом, чтобы при объявлении создавался новый объект класса **BankAccount:**

```
class CreateAccount
{
      ...
      static BankAccount NewBankAccount( )
      {
            BankAccount created = new BankAccount( );
            ...
            created.accNo = number;
            ...
      }
      ...
}
```

- Сохраните проект.
- Откомпилируйте и запустите программу. Убедитесь в том, что данные, введенные пользователем и считанные с консоли, корректно выводятся на экран с помощью метода **CreateAccount.Write**.

➢ **Инкапсулируйте данные класса BankAccount**

- В настоящий момент все члены класса **BankAccount**, описывающие данные, определены как **public**. Сделайте их **private**:

```
class BankAccount
{
      private long accNo;
      private decimal accBal;
```

```
        private AccountType accType;
    }
```

- Откомпилируйте программу. Вы получите сообщение об ошибке. Ошибка происходит в классе **CreateAccount**:

```
class CreateAccount
{
      ...
      static BankAccount NewBankAccount( )
      {
            BankAccount created = new BankAccount( );
            ...
            created.accNo = number; // Error here again
            ...
      }
      ...
}
```

- На этот раз не проходит компиляция инициализации членов класса, описывающих данные, т.к. они определены как **private**. Только методы класса **BankAccount** имеют доступ к **private** данным этого класса. Вам необходимо написать для класса **BankAccount public** метод, который будет производить инициализацию его членов. Сделайте следующее. В класс **BankAccount** добавьте нестатический **public** метод **Populate**. Это будет **void-**метод с двумя параметрами: первый типа **long** (номер банковского счета), второй типа **decimal** (баланс счета). В теле этого метода значение **long**-параметра будет присваиваться полю *accNo*, а значение decimal-параметра полю *accBal***.** Здесь же полю *accType* будет присваиваться значение **AccountType.Checking**.

```
class BankAccount
{
      public void Populate(long number, decimal balance)
      {
            accNo = number;
            accBal = balance;
            accType = AccountType.Checking;
      }
      private long accNo;
      private decimal accBal;
      private AccountType accType;
}
```

- В методе **CreateAccount.NewbankAccount** закомментируйте три выражения, в которых присваиваются значения переменной *created*. Вместо них добавьте вызов метода **Populate** для переменной *created*, передав ему в качестве аргументов переменные **number** и **balance**:

```
class CreateAccount
{
      ...
      static BankAccount NewBankAccount( )
      {
            BankAccount created = new BankAccount( );
            ...
            // created.accNo = number;
```

```
                // created.accBal = balance;
                // created.accType = AccountType.Checking;

                created.Populate(number, balance);
                ...
            }
            ...
    }
```

- Сохраните проект.
- Откомпилируйте программу. Вы получите сообщение об ошибке. В методе **CreateAccount.Write** осталось три выражения, в которых производится попытка напрямую обратиться к **private** данным класса **BankAccount**. Вам необходимо написать для класса **BankAccount** три **public** метода, которые будут возвращать значение этих трех полей класса. Сделайте следующее:
  - В класс **BankAccount** добавьте нестатический **public** метод **Number.** Этот метод будет возвращать значение типа **long** и не будет использовать параметры. Он будет возвращать значение поля *accNo*:

```
class BankAccount
{
  public void Populate(...) ...

  public long Number( )
  {
   return accNo;
  }
  ...
}
```

  - В класс **BankAccount** добавьте нестатический **public** метод **Balance.** Этот метод будет возвращать значение типа **decimal** и не будет использовать параметры. Он будет возвращать значение поля *accBal*:

```
class BankAccount
{
  public void Populate(...) ...

  ...
  public decimal Balance( )
  {
   return accBal;
  }
  ...
}
```

  - В класс **BankAccount** добавьте нестатический **public** метод **Type.** Этот метод будет возвращать значение типа **AccountType** и не будет использовать параметры. Он будет возвращать значение поля *accType*:

```
class BankAccount
{
  public void Populate(...) ...
  ...
```

```
        public AccountType Type( )
        {
         return accType;
        }
        ...
    }
```

- И, наконец, замените три выражения метода **CreateAccount.Write**, в которых производится попытка напрямую обратиться к **private** данным класса **BankAccount**, на три вызова только что созданных вами **public** методов:

```
class CreateAccount
{
    ...
    static void Write(BankAccount toWrite)
    {
         Console.WriteLine("Account number is {0}",
     toWrite.Number( ));
         Console.WriteLine("Account balance is {0}",
     toWrite.Balance( ));
         Console.WriteLine("Account type is {0}",
     toWrite.Type( ));
    }
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что введенные Вами данные корректно обрабатываются методом **BankAccount.Populate** и выводятся на экран методом **CreateAccount.Write**.

➢ **Продолжите инкапсуляцию данных класса BankAccount**

- Измените метод **BankAccount.Type** таким образом, что тип возвращаемых им значений стал **string**, вместо **AccountType**:

```
class BankAccount
{
        ...
        public string Type( )
        {
         return accType.ToString( );
        }
        ...
        private AccountType accType;
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что введенные вами данные корректно обрабатываются методом **BankAccount.Populate** и выводятся на экран методом **CreateAccount.Write**.

Упражнение 2. Генерация номеров банковских счетов

В этом упражнении Вы измените класс **BankAccount**, созданный в упражнении 1, таким образом, что он будет генерировать уникальные значения банковских счетов. Вы осуществите это через статическую переменную класса **BankAccount** и метод, который будет инкрементировать и возвращать значение этой переменной. Далее вы протестируете его работу, создав новый экземпляр банковского счета и вызвав для него метод, генерирующий номер банковского счета.

- **Сделайте номер каждого банковского счета уникальным**
  - Откройте проект UniqueNumbers.sln из папки *install folder*\Labs\Lab07\Starter\UniqueNumbers.

    Замечание: Этот проект совпадает с окончательным вариантом проекта упражнения 1.
  - В класс **BankAccount** добавьте статическую **private** переменную *nextAccNo* типа **long:**

```
class BankAccount
{
      ...
      private long accNo;
      private decimal accBal;
      private AccountType accType;

      private static long nextAccNo;
}
```

  - В класс **BankAccount** добавьте статический **public** метод **NextNumber.** Этот метод будет возвращать значение типа **long** и не будет использовать параметры. Он будет возвращать инкрементированное значение поля *nextAccNo*:

```
class BankAccount
{
      ...
      public static long NextNumber( )
      {
       return nextAccNo++;
      }

      private long accNo;
      private decimal accBal;
      private AccountType accType;
      private static long nextAccNo;
}
```

  - В методе **CreateAccount.NewBankAccount** закомментируйте выражение, запрашивающее у пользователя номер банковского счета:

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

  - В методе **CreateAccount.NewBankAccount** замените инициализацию переменной *number* вызовом только что созданного вами метода **BankAccount.NextNumber**:

```
//long number = long.Parse(Console.ReadLine( ));
long number = BankAccount.NextNumber( );
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что два банковских счета имеют номера 0 и1.
- В настоящий момент статическое поле **BankAccount.nextAccNo** по умолчанию инициализируется нулем. Проинициализируйте его явным образом значением 123.
- Откомпилируйте программу. Запустите программу. Убедитесь в том, что два банковских счета имеют номера 123 и124.

➢ **Продолжите инкапсуляцию данных класса BankAccount**

- Измените метод **BankAccount.Populate** таким образом, чтобы он принимал только один параметр – *balance* типа **decimal**. Внутри метода присвойте полю *accNo* значение, возвращаемое статическим методом **BankAccount.NextNumber**:

```
class BankAccount
{
      public void Populate(decimal balance)
      {
            accNo = NextNumber( );
            accBal = balance;
            accType = AccountType.Checking;
      }
      ...
}
```

- Измените модификатор доступа метода **BankAccount.NextNumber** на **private**:

```
class BankAccount
{
 ...
 private static long NextNumber( ) ...
}
```

- В методе **CreateAccount.NewBankAccount** закомментируйте объявление и инициализацию переменной *number*. Измените вызов метода **created.Populate** таким образом, чтобы он принимал только один параметр:

```
class CreateAccount
{
      ...
      static BankAccount NewBankAccount( )
      {
            BankAccount created = new BankAccount( );

            //long number = BankAccount.NextNumber( );
            ...
            created.Populate(balance);
            ...
      }
      ...
```

```
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что значения генерируемых номеров банковских счетов так и остались 123 и124.

Упражнение 3. Добавление дополнительных public методов

В этом упражнении Вы добавите еще два метода в класс **Account**: методы **Withdraw** и **Deposit**.

Метод **Withdraw** принимает на вход один параметр типа **decimal** и вычитает указанную сумму из баланса счета. Однако перед этим производится проверка, имеется ли на счету необходимая сумма, чтобы не превысить кредитный лимит счета. Метод возвращает значение типа **bool**, указывающее удалось ли снять со счета необходимую сумму.

Метод **Deposit** также принимает на вход один параметр типа **decimal** и прибавляет указанное значение к балансу счета. Метод возвращает новое значение баланса счета.

➢ **Добавьте метод Deposit в класс BankAccount**

- Откройте проект MoreMethods.sln из папки *install folder*\Labs\Lab07\Starter\MoreMethods.

  Замечание: Этот проект совпадает с окончательным вариантом проекта упражнения 2.
- В класс **BankAccount** добавьте нестатический **public** метод **Deposit.** Этот метод принимает на вход параметр типа **decimal**, значение которого прибавляется к балансу счета. Метод возвращает новое значение баланса счета.

```
class BankAccount
{
      ...
      public decimal Deposit(decimal amount)
      {
            accBal += amount;
            return accBal;
      }
      ...
}
```

- В класс **CreateAccount** добавьте статический **public** метод **TestDeposit**. Это будет **void-**метод с одним параметром типа **BankAccount**. В теле метода у пользователя будет запрашиваться сумма, которую необходимо положить на счет. Это значение будет конвертироваться в тип **decimal** и передаваться в качестве входного параметра при вызове метода **Deposit**:

```
class CreateAccount
{
      ...
      public static void TestDeposit(BankAccount acc)
      {
```

```
            Console.Write("Enter amount to deposit: ");
         decimal amount = decimal.Parse(Console.ReadLine());
            acc.Deposit(amount);
        }
        ...
    }
```

- В метод **CreateAccount.Main** добавьте инструкции для вызова только что созданного вами метод **TestDeposit**. Убедитесь в том, что вы вызываете метод **TestDeposit** для обоих объектов банковских счетов. Для вывода информации о счете после осуществления вклада используйте метод **CreateAccount.Write**:

```
class CreateAccount
{
      static void Main( )
      {
            BankAccount berts = NewBankAccount( );
            Write(berts);
            TestDeposit(berts);
            Write(berts);

            BankAccount freds = NewBankAccount( );
            Write(freds);
            TestDeposit(freds);
            Write(freds);
      }
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что метод **Deposit** работает корректно.

  Замечание: Если у вас осталось время, вы можете добавить в тело метода **Deposit** проверку того, что передаваемое на вход значение параметра является неотрицательным.

➢ **Добавьте метод Withdraw в класс BankAccount**

- В класс **BankAccount** добавьте нестатический **public** метод **Withdraw.** Этот метод принимает на вход один параметр типа **decimal** и вычитает указанную сумму из баланса счета. Однако перед этим производится проверка, имеется ли на счету необходимая сумма, чтобы не превысить кредитный лимит счета. Метод возвращает значение типа **bool**, указывающее удалось ли снять со счета необходимую сумму.

```
class BankAccount
{
      ...
      public bool Withdraw(decimal amount)
      {
            bool sufficientFunds = accBal >= amount;
            if (sufficientFunds) {
```

```
                accBal -= amount;
            }
            return sufficientFunds;
        }
        ...
    }
```

- В класс **CreateAccount** добавьте статический **public** метод **TestWithdraw**. Это будет **void-**метод с одним параметром типа **BankAccount**. В теле метода у пользователя будет запрашиваться сумма, которую необходимо снять со счета. Это значение будет конвертироваться в тип **decimal** и передаваться в качестве входного параметра при вызове метода **Withdraw**. При неудачной попытке снятия денег со счета на экран будет выводиться сообщение:

```
class CreateAccount
{
        ...
        public static void TestWithdraw(BankAccount acc)
        {
            Console.Write("Enter amount to withdraw: ");
            decimal                amount                =
         decimal.Parse(Console.ReadLine());
            if (!acc.Withdraw(amount)) {
             Console.WriteLine("Insufficient funds.");
            }
        }
        ...
}
```

- В метод **CreateAccount.Main** добавьте инструкции для вызова только что созданного вами метод **TestWithdraw.** Убедитесь в том, что вы вызываете метод **TestWithdraw** для обоих объектов банковских счетов. Для вывода информации о счете после изъятия суммы со счета используйте метод **CreateAccount.Write**:

```
class CreateAccount
{
        static void Main( )
        {
            BankAccount berts = NewBankAccount( );
            Write(berts);
            TestDeposit(berts);
            Write(berts);
            TestWithdraw(berts);
            Write(berts);

            BankAccount freds = NewBankAccount( );
            Write(freds);
            TestDeposit(freds);
            Write(freds);
            TestWithdraw(freds);
            Write(freds);
        }
}
```

- Сохраните проект.

- Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу. Убедитесь в том, что метод **Withdraw** работает корректно. Протестируйте удачные и неудачные попытки снятия денег со счета.

Лабораторная работа 7. Создание и использование ссылочных переменных

Цель работы

Изучение ссылочных типов данных и приобретение навыков работы с интерфейсами.

Упражнение 1. Добавление метода экземпляра с двумя параметрами.

В лабораторной работе 6 вы разработали класс **BankAccount**. В этом упражнении Вы добавите в этот класс метод экземпляра **TransferFrom**, который будет переводить деньги с указанного счета на данный. Если вы не закончили выполнение лабораторной работы 6, то вы можете использовать класс **BankAccount**, находящийся в папке *install folder*\Labs\Lab08\Starter\Bank.

➢ **Создайте метод TransferFrom**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab08\Starter\Bank.
- Отредактируйте класс **BankAccount** следующим образом:
  - В класс **BankAccount** добавьте **public** метод экземпляра **TransferFrom.**
  - Первым параметром этого метода будет ссылка на другой объект класса банковского счета **BankAccount** с именем **accFrom**, с которого будут переводиться деньги.
  - Второй параметр *amount* типа **decimal** передается по значению и указывает количество денег, которые необходимо перевести.
  - Метод не возвращает значения.
- В тело метода **TransferFrom** добавьте две инструкции, выполняющие следующие действия:
  - С помощью метода **Withdraw** снимите со счета **accFrom** сумму, определенную параметром *amount***.**
  - Если метод **Withdraw** отработал успешно (удалось снять необходимую сумму), то положите эти деньги на текущий счет с помощью метода **Deposit**.

```
class BankAccount
{

      //As before…

public void TransferFrom(BankAccount accFrom, decimal amount)
      {
            if (accFrom.Withdraw(amount))
```

```
                this.Deposit(amount);
            }
        }
```

- **Протестируйте метод TransferFrom**
  - В текущий проект добавьте файл Test.cs.
  - В этот файл добавьте следующий тестирующий класс:

```
using System;

public class Test
{
      public static void Main()
      {
      }
}
```

  - В методе **Main** создайте два объекта типа **BankAccount** с начальным балансом $100 (используйте метод **Populate**).
  - Добавьте код, выводящий на экран тип, номер счета и текущий баланс обоих счетов.
  - Вызовете метод **TransferFrom** и переведите с одного счета на другой $10.
  - Добавьте код, выводящий на экран текущие балансы обоих счетов после перевода денег.

    Текст класса **Test** должен выглядеть следующим образом:

```
public static void Main( )
{
      BankAccount b1 = new BankAccount( );
      b1.Populate(100);

      BankAccount b2 = new BankAccount( );
      b2.Populate(100);

      Console.WriteLine("Before transfer");
      Console.WriteLine("{0} {1} {2}", b1.Type( ),
 b1.Number( ), b1.Balance( ));
      Console.WriteLine("{0} {1} {2}", b2.Type( ),
 b2.Number( ), b2.Balance( ));

      b1.TransferFrom(b2, 10);

      Console.WriteLine("After transfer");
      Console.WriteLine("{0} {1} {2}", b1.Type( ),
 b1.Number( ), b1.Balance( ));
      Console.WriteLine("{0} {1} {2}", b2.Type( ),
 b2.Number( ), b2.Balance( ));
}
```

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

Упражнение 2. Обращение строки

В лабораторной работе 4 Вы создали класс **Utils**, для которого разработали несколько полезных методов.

В этом упражнении вы добавите в класс **Utils** новый статический метод **Reverse,** который будет принимать строку и возвращать новую строку, символы которой будут выстроены в обратном порядке.

➢ **Создайте метод Reverse**

- Откройте проект Utils.sln из папки *install folder*\Labs\Lab08\Starter\Utils.
- В класс **Utils** добавьте статический **public** метод **Reverse** следующим образом:
  - Данный метод имеет один параметр *s*, являющийся ссылкой на тип **string**.
  - Тип возвращаемого значения **void**.
- В теле метода **Reverse** создайте переменную *sRev* типа **string**, в которой будет сохраняться результат. Проинициализируйте ее значением “”.
- Обращение строки реализуйте следующим образом:
  - Создайте цикл, в котором будет извлекаться по одному символу из строки *s*. Начните с конца строки (используйте свойство **Length**) и повторяйте эту операция до тех пор, пока не достигнете начала строки. Для обращения к отдельному символу строки можно использовать [ ].

    Замечание: Последний символ строки находится на позиции **Length-1** (отсчет ведется с 0).
  - Присоедините выделенный символ к концу строки *sRev*.

```
class Utils
{

   //As before
   // Reverse a string

   public static void Reverse(ref string s)
   {
         string sRev = "";

         for (int k = s.Length - 1; k >= 0 ; k--)
          sRev = sRev + s[k];

         // Return result to caller
         s = sRev;
   }
}
```

➢ **Протестируйте метод Reverse**

- В текущий проект добавьте файл Test.cs.
- В этот файл добавьте следующий тестирующий класс:

```
using System;

public class Test
{
      public static void Main()
      {
      }
}
```

- В методе **Main** создайте переменную типа **string**.
- Считайте значение в эту переменную с помощью метода **Console.ReadLine**.
- Передайте эту строку в качестве параметра в метод **Reverse.** Не забудьте ключевое слово **ref**.
- Выведите на экран значение, возвращаемое методом **Reverse.**

Текст класса **Test** может выглядеть следующим образом:

```
static void Main( )
{
      string message;

      // Get an input string
      Console.WriteLine("Enter string to reverse:");
      message = Console.ReadLine( );

      // Reverse the string
      Utils.Reverse(ref message);

      // Display the result
      Console.WriteLine(message);
}
```

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

Упражнение 3. Создание прописной версии текста файла

В этом упражнении Вы напишите программу, запрашивающую у пользователя имя текстового файла. Программа проверит, что указанный файл существует и выведет сообщение об ошибке и закроется, если это не так. Далее файл откроется и его содержимое будет скопировано в другой файл (пользователю будет предложено дать ему имя), но каждый символ будет переведен в верхний регистр.

Перед тем как приступить к выполнению задания рекомендуется ознакомиться с документацией по работе с пространством имен **System.IO** (.NET Framework SDK Help). Обратите особое внимание на классы **StreamReader** и **StreamWriter**.

➢ **Создайте приложение, копирующее содержимое файла.**

- Откройте проект CopyFileUpper.sln из папки *install folder*\Labs\Lab08\Starter\CopyFileUpper.

- Отредактируйте класс **CopyFileUpper** и добавьте инструкцию **using** для пространства имен **System.IO**.
- В методе **Main** объявите две переменные *sFrom* и *sTo* типа **string**, для хранения имен входного и выходного файлов.
- Объявите переменную *srForm* типа **StreamReader.** В этой переменной будет храниться ссылка на входной файл.
- Объявите переменную *swTo* типа **StreamWriter.** В этой переменной будет храниться ссылка на выходной файл.
- Запросите у пользователя имя входного файла, считайте его и сохраните в переменной *sFrom.*
- Запросите у пользователя имя выходного файла, считайте его и сохраните в переменной *sTo.*
- Операции ввода-вывода, которые вы будете производить, могут привести к выбросу исключений, поэтому создайте **try-catch** блок, который будет отлавливать исключения типа **FileNotFoundException** (для несуществующих файлов) и **Exception** (для всех остальных видов исключений). Для каждого из типов исключений выведите на экран осмысленное сообщение.
- В блоке **try** создайте новый объект типа **StreamReader**, используя имя входного файла, хранимое в *sFrom,* и сохраните его в ссылочной переменной *srForm* типа **StreamReader.**
- По аналогии создайте новый объект типа **StreamWriter**, используя имя выходного файла, хранимое в *sTo,* и сохраните его в ссылочной переменной *swTo* типа **StreamWriter.**
- Добавьте цикл **while,** который будет работать до тех пор, пока метод **Peek** не вернет значение -1. Внутри цикла:
  - С помощью метода **ReadLine** считывайте по одной строке из входного потока и сохраняйте их в переменной *sBuffer* типа **string**.
  - К переменной *sBuffer* применяйте метод **ToUpper**.
  - С помощью метода **WriteLine** отправляйте переменную *sBuffer* в выходной поток.
- После завершения работы цикла закройте входной и выходной потоки.

Текст файла CopyFileUpper.cs должен выглядеть следующим образом:

```
using System;
using System.IO;

class CopyFileUpper
{
      static void Main( )
      {
            string sFrom, sTo;
            StreamReader srFrom;
            StreamWriter swTo;
```

```
                    // Prompt for input file name
                    Console.Write("Copy from:");
                    sFrom = Console.ReadLine( );

                    // Prompt for output file name
                    Console.Write("Copy to:");
                    sTo = Console.ReadLine( );

  Console.WriteLine("Copy from {0} to {1}", sFrom, sTo);
                    try
                    {
                          srFrom = new StreamReader(sFrom);
                          swTo = new StreamWriter(sTo);

                          while (srFrom.Peek( )!=-1)
                          {
                                string sBuffer = srFrom.ReadLine( );
                                sBuffer = sBuffer.ToUpper( );
                                swTo.WriteLine(sBuffer);
                          }
                          swTo.Close( );
                          srFrom.Close( );

                }
                catch (FileNotFoundException)
                    {
                          Console.WriteLine("Input file not found");
                    }
                    catch (Exception e)
                    {
                     Console.WriteLine("Unexpected exception");
                     Console.WriteLine(e.ToString( ));
                    }
              }
        }
```

- Сохраните проект.
- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Протестируйте работу программы**

- Откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab08\Starter\CopyFileUpper\bin\debug.
- Запустите CopyFileUpper.
- При появлении подсказки укажите имя файла:
  *drive:**path*\CopyFileUpper.cs
  Это тот файл, который вы только что создали.
- Укажите имя выходного файла **Test.cs**.
- После завершения работы программы, откройте файл **Test.cs** с помощью любого текстового редактора. В нем должна содержаться прописная версия написанного вами кода.

Упражнение 4. Проверка наличия реализации интерфейса

В этом упражнении Вы добавите в класс **Utils** статический метод **IsItFormattable**, который принимает один параметр типа **object** и проверяет, реализует ли данный параметр интерфейс **System.IFormattable**. Если да, то возвращается значение **true**, если нет, то возвращается значение **false**.

Вы напишите тестовый код, в котором будет вызываться метод **Utils.IsItFormattable** с аргументами различного типа, а результаты будут выводиться на экран.

- **Создайте метод IsItFormattable**
  - Откройте проект InterfaceTest.sln из папки *install folder*\Labs\Lab08\Starter\InterfaceTest.
  - Отредактируйте класс **Utils** следующим образом:
    - Создайте статический **public** метод **IsItFormattable.**
    - Этот метод принимает один параметр *x* типа **object**, передаваемый по значению. Метод возвращает значения типа **bool**.
    - Чтобы определить, поддерживает ли передаваемый объект интерфейс **System.IFormattable** используйте оператор **is**. Если да, то метод возвращает значение **true**, если нет, то **false**.

      Итоговый текст метода должен выглядеть следующим образом:

```
using System;
...
class Utils
{
      public static bool IsItFormattable(object x)
      {
            // Use the is operator to test whether the
            // object has the IFormattable interface

            if (x is IFormattable)
                  return true;
            else
                  return false;
      }
}
```

- **Протестируйте метод IsItFormattable**
  - Отредактируйте файл класса **Test**.
  - В методе **Main** объявите и проинициализируйте переменные типов **int, ulong** и **string**.
  - Передайте каждую из переменных в качестве параметра в метод **Utils.IsItFormattable** и выведите результат каждого вызова на экран.
  - Текст класса **Test** может выглядеть следующим образом:

```
using System;
public class Test
{
      static void Main( )
```

```
        {
            int i = 0;
            ulong ul = 0;
            string s = "Test";

            Console.WriteLine("int: {0}",
         Utils.IsItFormattable(i));
            Console.WriteLine("ulong: {0}",
         Utils.IsItFormattable(ul));
            Console.WriteLine("String: {0}",
         Utils.IsItFormattable(s));
        }
    }
```

- Откомпилируйте и протестируйте код. Вы должны увидеть **true** для значений **int** и **ulong,** и **false** для **string**.

Упражнение 5. Работа с интерфейсами

В этом упражнении Вы создадите метод **Display**, который будет использовать оператор **as** для того, чтобы определить, поддерживает ли переданный в качестве параметра объект пользовательский интерфейс **IPrintable**. Если да, то будет вызываться метод этого интерфейса.

➢ **Создайте метод Display**

- Откройте проект TestDisplay.sln из папки *install folder*\Labs\Lab08\Starter\TestDisplay.

  В стартовом коде определен интерфейс **IPrintable,** который содержит метод **Print**. Класс, реализующий этот интерфейс, должен содержать метод **Print** для вывода на консоль значений, хранимых в объекте. В стартовом коде также определен класс **Coordinate**, реализующий интерфейс **IPrintable.** Объект типа **Coordinate,** содержит пару числовых значений, определяющих положение в двумерном пространстве.
- Отредактируйте класс **Utils** следующим образом:
  - Добавьте статический **public void-**метод **Display.** Этот метод принимает один параметр *item* типа **object**, передаваемый по значению.
  - В теле метода **Display** объявите интерфейсную переменную *ip* типа **IPrintable.**
  - Преобразуйте ссылку, содержащуюся в параметре *item,* в ссылку на интерфейс **IPrintable,** используя оператор **as**. Результат сохраните в *ip*.
  - Если значение *ip* не **null,** вызовите интерфейсный метод **Print.** Если оно **null**, то объект не поддерживает интерфейс **IPrintable**. В этом случае используйте метод **Console.WriteLine** для вывода на экран результата применения к параметру метода **ToString.**

    Итоговый текст метода должен выглядеть следующим образом:

```
public static void Display(object item)
{
```

```
        IPrintable ip;

        ip = (item as IPrintable);

        if (ip != null)
             ip.Print( );
        else
             Console.WriteLine(item.ToString( ));
    }
```

- **Протестируйте метод Display**
  - В методе **Main** класса **Test** создайте переменные типов **int, string** и **Coordinate.** Для инициализации переменной типа **Coordinate** вы можете использовать конструктор с двумя параметрами:
    ```
    Coordinate c = new Coordinate(21.0, 68.0);
    ```
  - Передайте по очереди каждую из переменных в качестве параметра в метод **Utils.Display,** чтобы вывести их значения на экран:
    ```
    public class Test
    {
       static void Main( )
       {
            int num = 65;
            string msg = "A String";
            Coordinate c = new Coordinate(21.0,68.0);

            Utils.Display(num);
            Utils.Display(msg);
            Utils.Display(c);
       }
    }
    ```
  - Откомпилируйте и протестируйте ваше приложение.

Лабораторная работа 8. Создание объектов и управление ресурсами

Цель работы

Изучение конструкторов при создании объектов и приобретение навыков работы с шаблонами для удаления объектов.

Упражнение 1. Разработка конструкторов

В этом упражнении Вы модифицируете класс **BankAccount**, созданный в предыдущих лабораторных работах. Вы удалите методы, генерирующие номер и тип счета и замените их набором конструкторов, которые могут использоваться при создании экземпляра класса **BankAccount.**

Вы переопределите конструктор по умолчанию, генерируя номера счета (тем же способом, что и раньше), задавая тип счета **Checking** и баланс, равный нулю.

Вы также создадите еще три конструктора, использующие различные комбинации параметров:

- Первый конструктор будет принимать значение типа **AccountType**. Он будет генерировать номер счета, устанавливать нулевой баланс, а тип счета будет устанавливаться равным значению, передаваемому в качестве аргумента.
- Второй конструктор будет принимать значение типа **decimal**. Он будет генерировать номер счета, задавать тип счета **Checking,** а баланс будет устанавливаться равным значению, передаваемому в качестве аргумента.
- Третий конструктор будет принимать значения типа **AccountType** и **decimal**. Он будет генерировать номер счета, тип счета будет устанавливаться равным значению, передаваемому в качестве аргумента типа **AccountType,** а баланс будет устанавливаться равным значению, передаваемому в качестве аргумента типа **decimal**

➢ **Создайте конструктор по умолчанию**

- Откройте проект Constructors.sln из папки *install folder*\Labs\\Lab09\Starter\Constructors.
- Удалите метод **Populate** из класса **BankAccount.**
- Создайте конструктор по умолчанию следующим образом:
  - Имя **BankAccount.**
  - Модификатор доступа **public**.
  - Без параметров.
  - Без типа возвращаемых значений.
  - В теле конструктора необходимо генерировать номер счета, используя метод **NextNumber**, задать тип счета **AccountType.Checking** и баланс счета, равный нулю.

  Итоговый текст конструктора должен выглядеть следующим образом:

```
public BankAccount( )
{
      accNo = NextNumber( );
      accType = AccountType.Checking;
      accBal = 0;
}
```

➢ **Создайте остальные конструкторы**

- Добавьте еще один конструктор, который будет принимать один параметр **aType** типа **AccountType**. В теле конструктора:
  - Сгенерируйте номер счета также как раньше.
  - Присвойте *accType* значение **aType.**
  - Присвойте *accBal* значение нуль.
- Добавьте еще один конструктор, который будет принимать один параметр **aBal** типа **decimal**. В теле конструктора:
  - Сгенерируйте номер счета.

- Присвойте *accType* значение **AccountType.Checking.**
- Присвойте *accBal* значение **aBal**.

- Добавьте последний конструктор, который будет принимать два параметра: **aType** типа **AccountType** и **aBal** типа **decimal**. В теле конструктора:
  - Сгенерируйте номер счета.
  - Присвойте *accType* значение **aType.**
  - Присвойте *accBal* значение **aBal**.

Итоговый текст всех трех конструкторов должен выглядеть следующим образом:

```
public BankAccount(AccountType aType)
{
      accNo = NextNumber( );
      accType = aType;
      accBal = 0;
}

public BankAccount(decimal aBal)
{
      accNo = NextNumber( );
      accType = AccountType.Checking;
      accBal = aBal;
}

public BankAccount(AccountType aType, decimal aBal)
{
      accNo = NextNumber( );
      accType = aType;
      accBal = aBal;
}
```

➢ **Протестируйте конструкторы**

- В методе **Main** класса **CreateAccount** объявите четыре переменных типа **BankAccount** с именами *acc1, acc2, acc3, acc4*.
- Проинициализируйте первую переменную с помощью конструктора по умолчанию.
- Проинициализируйте вторую переменную с помощью конструктора, принимающего один параметр типа **AccountType**. Для переменной *acc2* задайте тип счета **AccountType.Deposit**.
- Проинициализируйте третью переменную с помощью конструктора, принимающего один параметр типа **decimal**. Для переменной *acc3* задайте баланс счета равным 100.
- Проинициализируйте четвертую переменную с помощью конструктора, принимающего параметры типа **AccountType** и **decimal**. Для переменной *acc4* задайте тип счета **AccountType.Deposit** и баланс счета 500.
- Для вывода информации о каждом счете используйте метод **Write** класса **CreateAccount.**

Итоговый текст должен выглядеть следующим образом:

```
static void Main( )
{
      BankAccount acc1, acc2, acc3, acc4;

      acc1 = new BankAccount( );
      acc2 = new BankAccount(AccountType.Deposit);
      acc3 = new BankAccount(100);
      acc4 = new BankAccount(AccountType.Deposit, 500);

      Write(acc1);
      Write(acc2);
      Write(acc3);
      Write(acc4);
}
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо. Запустите программу и убедитесь в том, что выводятся корректные данные.

Упражнение 2. Инициализация данных только для чтения

В этом упражнении Вы создадите новый класс **BankTransaction.** В нем будет храниться информация обо всех операциях по снятию и добавления денег на счет.

При каждом изменении баланса счета с помощью методов **Deposit** и **Withdraw** будет создаваться новый объект типа **BankTransaction**, в котором будет храниться информация о текущей дате и времени (сгенерированная **System.DateTime**) и количестве денег снятых (отрицательное значение) или добавленных (положительное значение) на счет. Т.к. данные о транзакциях не могут изменяться после их создания, эта информация будет храниться в двух **readonly-**переменных объекта **BankTransaction.**

Конструктор объекта **BankTransaction** будет принимать один параметр типа **decimal**, который будет использоваться им для определения суммы денег, участвующих в транзакции. Дата и время будут генерироваться с помощью свойства **DateTime.Now** класса **System.DateTime,** возвращающего текущие дату и время.

Далее Вы измените класс **BankAccount** таким образом, чтобы создавать транзакции в методах **Deposit** и **Withdraw**. Транзакции будут храниться в объектной переменной типа **System.Collections.Queue** класса **BankAccount.** Очередь (queue) это структура, в которой храниться упорядоченный список объектов. У нее имеются методы для добавления элементов в очередь и выполнения итераций по элементам очереди. Использование очереди имеет преимущество по сравнению с использованием массива, т.к. ее размер не фиксирован. Она может увеличиваться по мере добавления новых транзакций.

➢ **Создайте класс BankTransaction**

- Откройте проект Constructors.sln из папки *Lab Files*\Lab09\Starter\Constructors, если он еще не открыт.
- Добавьте в проект новый класс с именем **BankTransaction.**
- В коде класса **BankTransaction** удалите директиву **namespace** вместе с открывающей и закрывающей фигурными скобками.
- Удалите конструктор по умолчанию, созданный Microsoft Visual Studio.
- Добавьте две **private readonly** объектных переменных:
  - *amount* типа **decimal**
  - *when* типа **DateTime.** Структура **System.DateTime** удобна для работы с датами и временем и содержит большое количество методов для работы с такими величинами.
- Добавьте методы **Amount** и **When**, которые будут возвращать значения двух объектных переменных:

```
private readonly decimal amount;
private readonly DateTime when;
...
public decimal Amount( )
{
 return amount;
}

public DateTime When( )
{
 return when;
}
```

➢ **Создайте конструктор**

- Для класса **BankTransaction** объявите **public** конструктор с одним параметром *tranAmount* типа **decimal**, который будет использоваться для задания значения объектной переменной *amount.*
- В теле конструктора присвойте переменной *when* значение **DateTime.Now.**

  Замечание: DateTime.Now – это свойство, а не метод, поэтому круглые скобки использовать не надо.

Итоговый текст конструктора должен выглядеть следующим образом:

```
public BankTransaction(decimal tranAmount)
{
      amount = tranAmount;
      when = DateTime.Now;
}
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Создайте транзакции**

- Как было сказано выше, транзакции будут создаваться в классе **BankAccount** и помещаться в очередь при каждом вызове методов **Deposit** и **Withdraw**. Вернитесь в класс **BankAccount**.

- Перед началом описания класса **BankAccount** добавьте следующую директиву **using**:

```
using System.Collections;
```

- В классе **BankAccount** объявите **private** переменную *tranQueue* типа **Queue** и проинициализируйте ее новой пустой очередью:

```
private Queue tranQueue = new Queue( );
```

- В методе **Deposit** перед инструкцией **return** создайте новую транзакцию, используя в качестве параметра сумму, вносимую на счет, и присоедините ее к очереди, используя метод **Enqueue**:

```
public decimal Deposit(decimal amount)
{
      accBal += amount;
      BankTransaction tran = new BankTransaction(amount);
      tranQueue.Enqueue(tran);
      return accBal;
}
```

- В методе **Withdraw,** при условии наличия на счете необходимой суммы, создайте новую транзакцию и присоедините ее к очереди *tranQueue*, также как и в методе **Deposit:**

```
public bool Withdraw(decimal amount)
{
      bool sufficientFunds = accBal >= amount;
      if (sufficientFunds) {
            accBal -= amount;
       BankTransaction tran = new BankTransaction(-amount);
            tranQueue.Enqueue(tran);
      }
      return sufficientFunds;
}
```

Замечание: Обратите внимание на то, что в методе **Withdraw** значением, передаваемым конструктору **BankTransaction,** является сумма, которую необходимо снять со счета, со знаком минус.

➢ **Протестируйте транзакции.**

- Для проведения тестирования в класс **BankAccount** добавьте метод **Transactions**, с типом возвращаемых значений **Queue.** Он будет возвращать значение *tranQueue*. Вы будете использовать этот метод на следующем шаге для вывода на экран информации о транзакциях. Метод должен выглядеть следующим образом:

```
public Queue Transactions( )
{
 return tranQueue;
}
```

- В классе **CreateAccount** измените метод **Write** таким образом, чтобы выводить информацию о транзакциях для каждого счета. Очереди реализуют интерфейс **IEnumerable**, а это значит, что для произведения итераций по элементам очереди вы можете использовать конструкцию **foreach**.

- В теле цикла **foreach** выведите на печать дату, время и сумму, участвующую в транзакции, используя методы **Amount** и **When:**

```
static void Write(BankAccount acc)
{
     Console.WriteLine("Account number is {0}",
 acc.Number( ));
     Console.WriteLine("Account balance is {0}",
 acc.Balance( ));
 Console.WriteLine("Account type is {0}", acc.Type( ));
     Console.WriteLine("Transactions:");
     foreach (BankTransaction tran in acc.Transactions( ))
     {
          Console.WriteLine("Date/Time: {0}\tAmount: {1}",
      tran.When( ), tran.Amount( ));
     }
     Console.WriteLine( );
}
```

- В метод **Main** добавьте инструкции для снятия и добавления денег на каждый из четырех счетов (*acc1, acc2, acc3, acc4*).
- Откомпилируйте проект и исправьте ошибки, если это необходимо.
- Запустите проект. Проверьте корректность выводимой информации о транзакциях.

Упражнение 3. Использование шаблона для удаления объектов

В этом упражнении Вы добавите в класс **BankAccount** метод **Dispose** для сохранения информации о транзакциях в файле Transaction.dat. Метод **Dispose** будет производить итерации по всем транзакциям, находящимся в очереди и сохранять информацию о транзакциях (журнал транзакций) в файле.

➢ **Создайте в классе BankAccount метод Dispose**

- Откройте проект Finalizers.sln из папки *install folder*\Labs\Lab09\Starter\Finalizers.
- Добавьте в объявлении класса **BankAccount** модификатор **sealed** и укажите наследование от интерфейса **IDisposable.** От класса, объявленного с ключевым словом **sealed** нельзя наследовать. Модификатор **sealed** добавлен для упрощения реализации метода **Dispose.**
- Добавьте объектную переменную *disposed* типа **bool** и присвойте ей значение **false**.
- В классе **BankAccount** объявите **public void-**метод **Dispose:**

```
public void Dispose( )
{

}
```

- В теле метода **Dispose** добавьте следующие инструкции:
  - Проверьте значение переменной *disposed*. Если оно равно **true,** то просто завершите работу метода и ничего не делайте.

- Если значение *disposed* **false,** создайте новую переменную типа **StreamWriter,** которая открывает файл Transaction.dat, находящийся в текущей директории в режиме дозаписи. Этого можно добиться, используя метод File.Append:

```
StreamWriter swFile =File.AppendText("Transactions.Dat");
```

- Для записи номера, типа и баланса счета используйте инструкцию **WriteLine:**

```
swFile.WriteLine("Account number is {0}", accNo);
swFile.WriteLine("Account balance is {0}", accBal);
swFile.WriteLine("Account type is {0}", accType);
```

- Произведите итерации по всем объектам типа **BankTransaction**, находящимся в очереди *tranQueue* и запишите сумму, дату и время транзакции. Используйте конструкцию **foreach** по аналогии с тем, что вы делали в предыдущих упражнениях.
- Закройте **StreamWriter.**
- Присвойте *disposed* значение **true.**
- Вызовите метод **GC.SuppressFinalize.**

Итоговый код метода **Dispose** должен выглядеть следующим образом:

```
public void Dispose( )
{
      if (!disposed)
      {
            StreamWriter swFile =
       File.AppendText("Transactions.Dat");
       swFile.WriteLine("Account number is {0}", accNo);
       swFile.WriteLine("Account balance is {0}", accBal);
       swFile.WriteLine("Account type is {0}", accType);
            swFile.WriteLine("Transactions:");
            foreach(BankTransaction tran in tranQueue)
            {
    swFile.WriteLine("Date/Time: {0}\tAmount:{1}", tran.When( ),
tran.Amount( ));
            }
            swFile.Close( );
            disposed = true;
            GC.SuppressFinalize(this);
      }
}
```

- В классе **BankAccount** добавьте деструктор, вызывающий метод **Dispose**.
- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Протестируйте деструктор**

- Откройте файл CreateAccount.cs,
- Измените код метода **Main**, используя инструкцию **using** следующим образом:

```
using (BankAccount acc1 = new BankAccount( ))
{
```

```
            acc1.Deposit(100);
            acc1.Withdraw(50);
            acc1.Deposit(75);
            acc1.Withdraw(50);
            acc1.Withdraw(30);
            acc1.Deposit(40);
            acc1.Deposit(200);
            acc1.Withdraw(250);
            acc1.Deposit(25);
            Write(acc1);
        }
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.
- Запустите программу.
- Откройте текстовый редактор и просмотрите содержимое файла, находящегося в папке *install* folder\Labs\Lab09\Starter\Finalizers\bin\Debug.

Лабораторная работа 9. Использование наследования при реализации интерфейсов

Цель работы

Изучение наследования как важного элемента объектно-ориентированного программирования и приобретение навыков реализации наследования на основе интерфейсов.

Упражнение 1. Преобразование исходного файла на C# в файл HTML

Оболочки (frameworks) очень полезны, так как представляют собой удобный, гибкий механизм написания кода. В отличие от библиотек, которые используются для прямого вызова методов, вы используете framework, для создания нового класса, реализующего интерфейс.

Код framework может полиморфно вызывать методы вашего класса через методы интерфейса. Таким образом, грамотно разработанный framework может использоваться различными способами, в то время как метод библиотеки может использоваться лишь одним способом.

В этом упражнении используется заранее созданная иерархия интерфейсов и классов, образующих миниатюрный framework. Framework снабжает метками каждый символ исходного файла кода на языке C# и хранит различные варианты меток в коллекции в классе **SourceFile**. Также существует интерфейс **ITokenVisitor** с набором методов **Visit**, который вместе с методом **Accept** класса **SourceFile** позволяет поочередно просмотреть каждую метку исходного файла. При просмотре метки ваш класс может выполнить для нее любой набор необходимых действий.

Абстрактный класс **NullTokenVisitor** был создан для реализации всех методов **Visit** интерфейса **ITokenVisitor** пустыми методами. Если Вы не хотите реализовывать каждый метод интерфейса **ITokenVisitor**, то Вы можете наследовать от класса **NullTokenVisitor** и реализовывать только те методы **Visit,** какие захотите.

В этом упражнении Вы унаследуете класс **HTMLTokenVisitor** от интерфейса **ITokenVisitor**.

В производном классе Вы реализуете каждый перегруженный метод **Visit** для вывода на консоль каждой метки и заключите ее в тэги HTML.

Вы запустите простой batch-файл, который запустит созданный Вами файл и перенаправит вывод на консоль, создав HTML-страницу на основе каскадной таблицы стилей. Далее откроете HTML-страницу в Microsoft Internet Explorer и увидите текст исходного файла с использованием цветного синтаксиса.

- **Ознакомьтесь с интерфейсами**
  - Откройте проект ColorToken.sln из папки *install folder*\Labs\Lab10\Starter\ColorTokeniser.
  - Изучите классы и интерфейсы в файлах Itoken.cs, Itoken_visitor.cs и source_file.cs. Они образуют следующую иерархию (рис. 9.1):

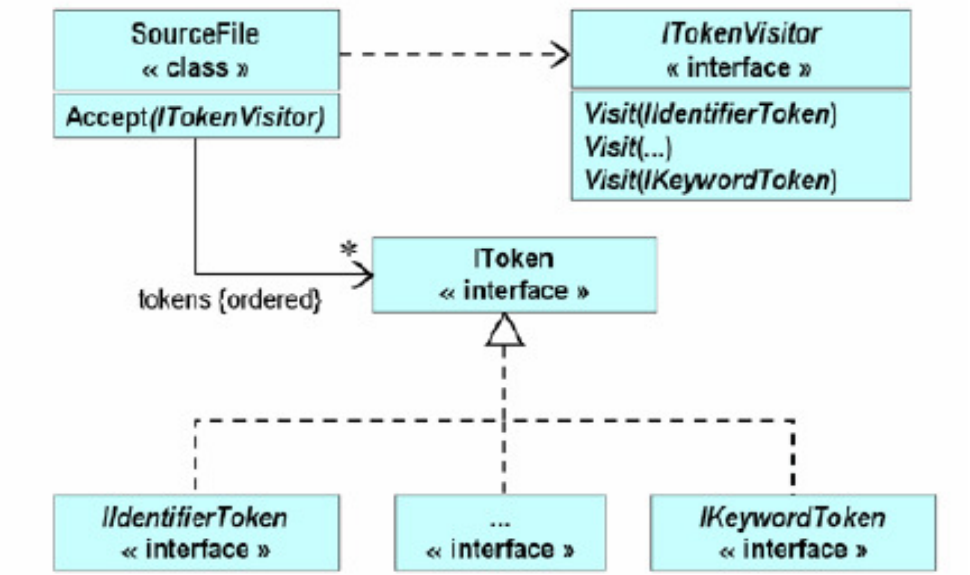

Рис. 9.1 Иерархия наследования

- **Создайте абстрактный класс NullTokenVisitor**
  - Откройте файл null_token_visitor.cs.

    Обратите внимание на то, что класс **NullTokenVisitor** наследуется от интерфейса **ITokenVisitor**, хотя в данный момент он не реализует ни один из методов интерфейса. Вы реализуете все унаследованные методы пустыми методами без тела, для того, чтобы появилась возможность постепенно разрабатывать класс **HTMLTokenVisitor.**
  - В класс **NullTokenVisitor** добавьте виртуальный **public void-**метод **Visit**, принимающий один параметр типа **ILineStartToken.** Тело метода останется пустым. Метод будет выглядеть следующим образом:

```
public class NullTokenVisitor : ITokenVisitor
{
```

```
        public virtual void Visit(ILineStartToken t) { }
        ...
    }
```

- Повторите предыдущие действия для всех вариантов перегруженного метода **Visit**, объявленных в интерфейсе **ITokenVisitor.** Тело каждого метода оставьте пустым.
- Сохраните проект.
- Откомпилируйте файл null_token_visitor.cs.

  Если Вы реализовали все методы **Visit** интерфейса **ITokenVisitor**, то компиляция пройдет успешно. Если нет, то компилятор выдаст сообщение об ошибке.
- В класс **NullTokenVisitor** добавьте статический **private void** метод **Test** без параметров. Тело этого метода состоит всего из одной инструкции, которая создает новый объект типа **NullTokenVisitor.** Эта инструкция позволит убедиться в том, что класс **NullTokenVisitor** реализует все методы **Visit** интерфейса **ITokenVisitor** и можно создавать его экземпляры.

  Текст этого метода должен выглядеть следующим образом:

```
public class NullTokenVisitor : ITokenVisitor
{
 ...
        static void Test( )
        {
         new NullTokenVisitor( );
        }
}
```

- Сохраните проект.
- Откомпилируйте файл null_token_visitor.cs и исправьте ошибки, если это необходимо.
- Измените определение класса **NullTokenVisitor.** Т.к. Вы собираетесь только наследовать от этого класса, а не создавать его объекты, Вам необходимо определить его как абстрактный.
- Снова откомпилируйте файл null_token_visitor.cs. Теперь использование оператора **new** в методе **Test** приводит к ошибке, т.к. Вы не можете создавать объекты абстрактного класса.
- Удалите метод **Test**.
- Класс **NullTokenVisitor** теперь должен выглядеть следующим образом:

```
public abstract class NullTokenVisitor : ITokenVisitor
    {
        public virtual void Visit(ILineStartToken t) { }
        public virtual void Visit(ILineEndToken t) { }

        public virtual void Visit(ICommentToken t) { }
        public virtual void Visit(IDirectiveToken t) { }
        public virtual void Visit(IIdentifierToken t) { }
        public virtual void Visit(IKeywordToken t) { }
        public virtual void Visit(IWhiteSpaceToken t) { }
```

```
        public virtual void Visit(IOtherToken t) { }
    }
```

➢ **Создайте класс HTMLTokenVisitor**

- Откройте файл html_token_visitor.cs.
- Измените класс **HTMLTokenVisitor** таким образом, чтобы он наследовался от абстрактного класса **NullTokenVisitor** (рис. 9.2)**.**

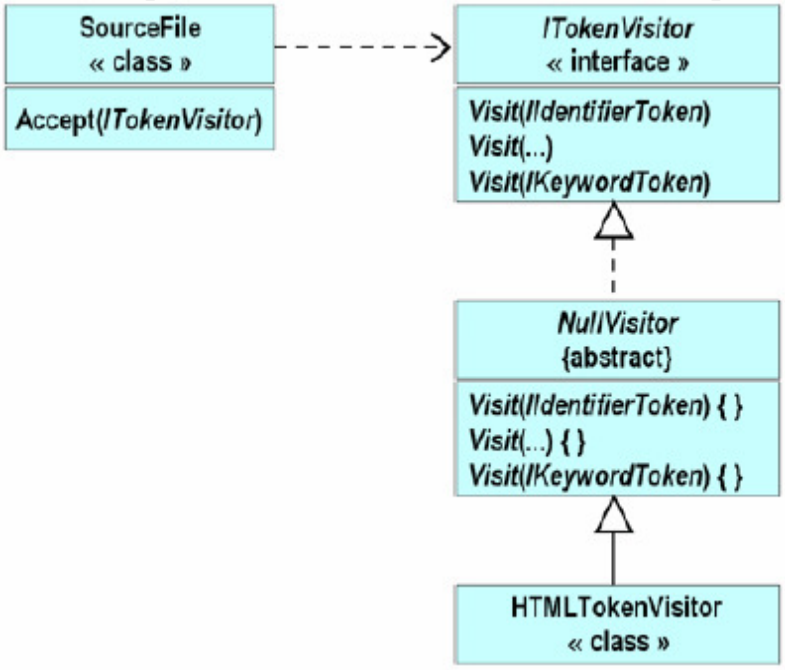

Рис.9.2 Реализация наследования

- Откройте файл main.cs и добавьте в статический метод **InnerMain** две инструкции.
  - В первой инструкции объявляется переменная *visitor* типа **HTMLTokenVisitor** и инициализируется новым объектом типа **HTMLTokenVisitor.**
  - Во второй инструкции переменная *visitor* передается в качестве параметра в метод **Accept,** вызванный для ранее объявленной переменной *source*.
- Сохраните проект
- Откомпилируйте программу и исправьте ошибки, если это необходимо.
- Запустите программу из командной строки, передав в качестве аргумента имя исходного .cs-файла, находящегося в папке *install folder*\Labs\Lab10\Starter\ColorTokeniser\bin\debug.

  Ничего не произойдет, т.к. Вы еще не определили методы класса **HTMLTokenVisitor.**
- В класс **HTMLTokenVisitor** добавьте нестатический **public void-**метод **Visit**, принимающий один параметр **line** типа **ILineStartToken**.

  В теле метода напишите одну инструкцию, в которой будет вызваться метод **Write** (*не* **WriteLine**), для вывода на экран значения

line.Number(). Обратите внимание на то, что **Number –** это метод, объявленный в интерфейсе **ILineStartToken.** Не определяйте этот метод как **virtual** или **override.** Итоговый текст метода:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
 public void Visit(ILineStartToken line)
 {
  Console.Write(line.Number( )); // Not WriteLine
 }
}
```

- Сохраните проект.

  Запустите программу так же, как раньше. Ничего не произойдет, т.к. метод **Visit** класса **HTMLTokenVisitor** скрывает метод **Visit** базового класса **NullTokenVisitor.**
- Измените метод **HTMLTokenVisitor.Visit(ILineStartToken)** таким образом, чтобы он переопределял метод **Visit** базового класса.

  Это сделает метод **HTMLTokenVisitor.Visit** полиморфным:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
 public override void Visit(ILineStartToken line)
 {
  Console.Write(line.Number( ));
 }
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо.

  Запустите программу так же, как раньше. Будут выведены числа в порядке возрастания без промежуточных пробелов (числа являются номерами строк указанного файла).
- В классе **HTMLTokenVisitor** создайте перегруженный нестатический **public void-**метод **Visit,** принимающий один параметр типа **ILineEndToken**.

  Этот метод добавляет новую строку между строками выводимых символов. Обратите внимание на то, что этот метод объявлен в интерфейсе **ITokenVisitor**. В теле метода добавьте инструкцию для вставки новой строки. (Используйте метод **WriteLine**, а не **Write**):

```
public class HTMLTokenVisitor : NullTokenVisitor
{
 ...
 public override void Visit(ILineEndToken t)
 {
  Console.WriteLine( ); // Not Write
 }
}
```

- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо.

Запустите программу так же, как раньше. На этот раз номер каждой строки исходного файла выведется в отдельной строке.

- **Используйте класс HTMLTokenVisitor для отображения кода исходного файла**
  - В класс **HTMLTokenVisitor** добавьте нестатический **public void-** метод **Visit**, принимающий один параметр **token** типа **IIdentifierToken**. Он должен переопределять соответствующий метод базового класса **NullTokenVisitor.**
  - В теле метода напишите одну инструкцию, в которой будет вызываться метод **Write** для вывода на экран значения **token,** приведенного к типу **string.** Итоговый текст метода:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      ...
      public override void Visit(IIdentifierToken token)
      {
       Console.Write(token.ToString( ));
      }
}
```

  Замечание: Откройте файл IToken.cs и обратите внимание на то, что интерфейс **IIdentifierToken** наследуется от интерфейса **IToken** в котором объявлен метод **ToString.**
  - Сохраните проект.
  - Откомпилируйте программу и исправьте ошибки, если это необходимо. Запустите программу так же, как раньше. На этот раз на экран выводятся все идентификаторы.
  - Повторите четыре предыдущих шага для создания в классе **HTMLTokenVisitor** еще четырех перегруженных методов **Visit.**

    Каждый из них будет принимать один параметр типа **ICommentToken, IKeywordToken, IWhiteSpaceToken** и **IOtherToken** соответственно. Тело этих методов будет точно такими же, как и у описанного выше метода **Visit(IIdentifierToken token)**.

- **Преобразуйте исходный файл на языке C# в файл HTML**
  - В папке *install folder*\Labs\Lab10\Starter\ColorTokeniser\bin\debug располагается batch-файл с именем generate.bat. Этот скрипт запускает программу ColorTokeniser, используя передаваемой Вами параметр командной строки. Он также производит предварительную и заключительную обработку передаваемого файла, используя каскадную таблицу стилей (code_style.css) для преобразования результатов в HTML.

    Из командной строки запустите программу, используя batch-файл, передав в качестве параметра файл token.cs. Передайте результаты в другой файл с расширением .html. Например:

```
generate token.cs > token.html
```

- Для просмотра созданного вами .html-файла воспользуйтесь Internet Explorer. Это можно сделать, набрав в командной строке **token.html**.

  В результате Вы увидите множество дефектов форматирования. Номера строк, большие 9, имеют отступ, отличный от номеров строк, меньших 10. Номера строк имеют тот же цвет, что и текст файла, что не очень удобно.

➢ **Найдите и исправьте проблемы, связанные с отступом номеров строк**

- Измените определение метода **Visit(ILineStartToken)** таким образом, чтобы исправить данную проблему:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(ILineStartToken line)
      {
            Console.Write("<span class=\"line_number\">");
            Console.Write("{0,3}", line.Number( ));
            Console.Write("</span>");
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer.

  Там все еще имеются некоторые дефекты. Сравните вид token.html в Internet Explorer с оригиналом token.cs. Обратите внимание на то, что первый комментарий в token.cs (///<summary>) отражается в браузере в виде “///”. Тэг <summary> утерян, т.к. в HTML некоторые символы имеют специальное назначение. Для отображения **<** в коде HTML необходимо использовать **<** для отображения **>** в коде HTML необходимо использовать **>** для отображения **&** необходимо использовать **&**.

➢ **Внесите необходимые изменения для корректного отображения угловых скобок и амперсандов**

- В **HTMLTokenVisitor** добавьте нестатический **private** метод **FilteredWrite,** возвращающий **void** и принимающий один параметр **token** типа **IToken**.

  Этот метод будет создавать из **token** строку **dst**, применяя к нему трансформации, описанные выше. Итоговый код должен выглядеть следующим образом:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      ...
      private void FilteredWrite(IToken token)
      {
```

```
                string src = token.ToString( );
                for (int i = 0; i != src.Length; i++) {
                      string dst;
                      switch (src[i]) {
                      case '<' :
                       dst = "&lt;"; break;
                      case '>' :
                       dst = "&gt;"; break;
                      case '&' :
                       dst = "&amp;"; break;
                      default :
                       dst = new string(src[i], 1); break;
                      }
                      Console.Write(dst);
                }
          }
    }
```

- Измените определение метода **HTMLTokenVisitor.Visit(ICommentToken).** Используйте вместо **Console.Write** метод **FilteredWrite:**

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(ICommentToken token)
      {
       FilteredWrite(token);
      }
      ...
}
```

- Измените определение метода **HTMLTokenVisitor.Visit(IOtherToken).** Используйте вместо **Console.Write** метод **FilteredWrite:**

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(IOtherToken token)
      {
       FilteredWrite(token);
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer и убедитесь в том, что угловые скобки и амперсанды теперь отображаются корректно.

➢ **Добавьте в HTML файл цветные комментарии**

- С помощью Notepad откройте таблицу стилей code_style.css из папки *install folder*\Labs\Lab10\Starter\ColorTokeniser\bin\debug.

Каскадная таблица стилей code_style.css будет использоваться для цветового оформления HTML файла. Этот файл был заранее создан для Вас. Вот часть его содержимого:

```
...
SPAN.LINE_NUMBER
{
      background-color: white;
      color: gray;
}
...
SPAN.COMMENT
{
      color: green;
      font-style: italic;
}
```

Метод **HTMLTokenVisitor.Visit(ILineStartToken)** уже использует эту таблицу стилей:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(ILineStartToken line)
      {
            Console.Write("<span class=\"line_number\">");
            Console.Write("{0,3}", line.Number( ));
            Console.Write("</span>");
      }
      ...
}
```

Обратите внимание на то, что в методе пишутся слова “span” и “line_number”, а в таблице стилей содержится элемент SPAN.LINE_NUMBER.

- Измените тело метода **HTMLTokenVisitor.Visit(ICommentToken)** следующим образом:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(ICommentToken token)
      {
            Console.Write("<span class=\"comment\">");
            FilteredWrite(token);
            Console.Write("</span>");
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer.

Убедитесь в том, что для комментариев к исходному файлу теперь используется зеленый шрифт и курсив

➢ **Добавьте в HTML файл цветные ключевые слова**

- Обратите внимание на то, что в файле code_style.css содержится следующий элемент:

```
...
SPAN.KEYWORD
{
 color: blue;
}
...
```

- Внесите изменения в тело метода **HTMLTokenVisitor.Visit(IKeywordToken),** чтобы использовать стиль, определенный в таблице стилей:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      public override void Visit(IKeywordToken token)
      {
            Console.Write("<span class=\"keyword\">");
            FilteredWrite(token);
            Console.Write("</span>");
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer и убедитесь в том, что ключевые слова теперь голубые.

➢ **Измените методы Visit для исключения дублирования кода**

- Обратите внимание на дублирование кода в двух предыдущих методах **Visit**.

  Вы можете внести изменения в методы **Visit** для предотвращения этого дублирования. Создайте новый нестатический **private** метод **SpannedFilteredWrite,** возвращающий **void** и принимающий два параметра: строку *spanName* и *token* типа **IToken**. Тело этого метода будет состоять из трех инструкций. Первая будет выводить строку с тэгом span, используя параметр *spanName*. Вторая инструкция вызывает метод **FilteredWrite,** передавая в него **token** в качестве аргумента. Третья инструкция будет выводить на консоль закрывающий тэг span. Итоговый код должен выглядеть следующим образом:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
...
      private  void  SpannedFilteredWrite(string  spanName,
 IToken token)
      {
      Console.Write("<span class=\"{0}\">", spanName);
      FilteredWrite(token);
```

```
            Console.Write("</span>");
            }
            ...
      }
```

- Перепишите метод **HTMLTokenVisitor.Visit(ICommentToken),** с использованием только что созданного метода:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      ...
      public override void Visit(ICommentToken token)
      {
            SpannedFilteredWrite("comment", token);
      }
      ...
}
```

- Перепишите метод **HTMLTokenVisitor.Visit(IKeywordToken),** с использованием только что созданного метода:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      ...
      public override void Visit(IKeywordToken token)
      {
       SpannedFilteredWrite("keyword", token);
      }
      ...
}
```

- Перепишите метод **HTMLTokenVisitor.Visit(IIdentifierToken),** с использованием метода **SpannedFilteredWrite**. Это необходимо сделать, потому что в файле code_style.css задано оформление для символов идентификаторов:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
      ...
      public override void Visit(IIdentifierToken token)
      {
       SpannedFilteredWrite("identifier", token);
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer и убедитесь в том, что комментарии остались зелеными, а ключевые слова голубыми.

➢ **Реализуйте HTMLTokenVisitor напрямую от ITokenVisitor**

- Откройте файл html_token_visitor.cs.
- Измените код таким образом, чтобы класс **HTMLTokenVisitor** наследовался от интерфейса **ITokenVisitor**. Т.к. Вы реализовали

почти все методы **Visit** в **HTMLTokenVisitor**, Вам больше нет необходимости наследовать от абстрактного класса **NullTokenVisitor** (в котором содержаться пустые реализации всех методов **ITokenVisitor**). Его можно наследовать напрямую интерфейса **ITokenVisitor.**

- Класс должен выглядеть следующим образом:

```
public class HTMLTokenVisitor : ITokenVisitor
{
  ...
}
```

- Сохраните проект.
- Откомпилируйте программу.

  Вы получите большое количество сообщений об ошибках. Дело в том, что в определениях методов **Visit** класса **HTMLTokenVisitor** осталось ключевое слово **override**. Но вы не можете переопределять методы интерфейса.
- Из определения каждого метода **Visit** удалите ключевое слово **override.**
- Откомпилируйте программу.

  Вы все равно получите сообщение об ошибке. Дело в том, что класс **HTMLTokenVisitor** не реализует метод **Visit(IDirectiveToken),** унаследованный от интерфейса **ITokenVisitor**. Ранее **HTMLTokenVisitor** наследовал пустую реализацию этого метода от **NullTokenVisitor.**
- В классе **HTMLTokenVisitor** создайте новый нестатический **public** метод **Visit,** возвращающий **void** и принимающий один параметр *token* типа **IDirectiveToken**. Это решит проблему, связанную с отсутствием реализации.

  В теле этого метода будет вызываться метод **SpannedFilteredWrite,** в который будут передаваться два параметра: строковый литерал “directive” и переменная *token*.

```
public class HTMLTokenVisitor : ITokenVisitor
{
      ...
      public void Visit(IDirectiveToken token)
      {
            SpannedFilteredWrite("directive", token);
      }
      ...
}
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer и убедитесь в том, что комментарии остались зелеными, а ключевые слова голубыми.

➢ **Предотвратите использование HTMLTokenVisitor в качестве базового класса**

- В объявлении класса **HTMLTokenVisitor** используйте ключевое слово **sealed**.

  Так как методы **HTMLTokenVisitor** перестали быть виртуальными, имеет смысл объявить класс **HTMLTokenVisitor** как **sealed.** Это показано в следующем коде:

```
public sealed class HTMLTokenVisitor : ITokenVisitor
{ ... }
```

- Сохраните проект.
- Запустите программу и исправьте ошибки, если это необходимо.
- Заново создайте файл token.html из файла token.cs из командной строки:

```
generate token.cs > token.html
```

- Откройте token.html в Internet Explorer и убедитесь в том, что комментарии остались зелеными, а ключевые слова голубыми.

Упражнение 2. Преобразование исходного файла на C# в файл HTML

В этом упражнении Вы исследуете другое приложение, в котором будет использоваться framework из упражнения 1.

В этом приложении класс **ColorTokenVisitor** наследуется от интерфейса **ITokenVisitor.** Методы **Visit** этого класса выводят цветные элементы в **RichTextBox** приложения Microsoft Windows Forms. Используемые классы образуют следующую иерархию (рис.9.3):

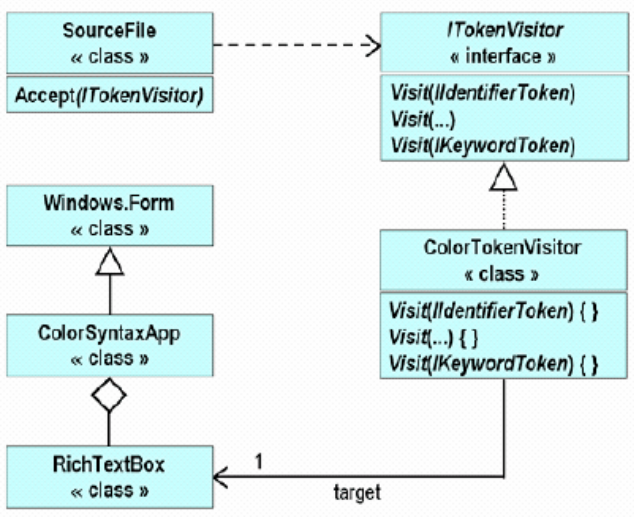

Рис. 9.3 Иерархия классов

➢ **Ознакомьтесь с интерфейсами**

- Откройте проект ColorSyntaxApp.sln из папки *install folder*\Labs\Lab10\Solution\ColourSyntaxApp
- Изучите содержимое двух .cs файлов. Обратите внимание на то, что класс **ColorTokenVisitor** очень похож на класс **HTMLTokenVisitor,** созданный Вами в упражнении 1. Главное различие заключается в том, что класс **ColorTokenVisitor** выводит цветные элементы в компонент формы **RichTextBox**, а не на консоль.

- Откомпилируйте проект.
- Запустите приложение.
    - Нажмите на кнопку **Open File.**
    - В появившемся диалоговом окне выберите исходный файл .cs.
    - Нажмите на кнопку **Open.**

    Появится цветное содержимое выбранного исходного .cs файла.

Лабораторная работа 10. Использование модификатора доступа internal и создание сборок

Цель работы

Изучение возможностей языка для логического группирования компонентов и приобретение навыков работы по реализации основного строительного блока любого приложения .NET, образующего базовую единицу развертывания.

Упражнение 1. Создание класса с модификатором internal

В этом упражнении Вы:

1. Создадите новый класс **Bank**, который будет использоваться как фабрика для создания объектов типа **BankAccount**.
2. Измените модификаторы доступа конструкторов **BankAccount** на internal.
3. В класс **Bank** добавите перегруженные методы **CreateAccount,** который могут использоваться клиентами для доступа к счетам и запросов на создание новых счетов.
4. Сделаете все методы класс **Bank** статическими (и **public**) и добавите в него закрытый конструктор для того, чтобы предотвратить возможность создания ненужных экземпляров класса **Bank.**
5. Будете хранить **BankAccount** в **Bank**, используя хэш-таблицу (**System.Collections.Hashtable**).
6. Протестируете работу класса **Bank.**

➢ **Создайте класс Bank**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab11\Exercise 1\Starter\Bank.
- В файле BankAccount.cs просмотрите четыре конструктора класса **BankAccount.**

  В классе **Bank** Вы создадите четыре перегруженных метода **CreateAccount**, которые будут вызывать каждый из этих конструкторов соответственно.

  Откройте файл Bank.cs и в классе **Bank** создайте нестатический **public** метод **CreateAccount** без параметров и возвращающий значение типа **BankAccount.** В теле этого метода должен возвращаться новый объект **BankAccount,** созданный через вызов конструктора **BankAccount** без параметров.

- В метод **Main** файла CreateAccount.cs добавьте следующие инструкции для тестирования метода **CreateAccount:**

```
Console.WriteLine("Sid's Account");
Bank bank = new Bank( );
BankAccount sids = bank.CreateAccount( );
TestDeposit(sids);
TestWithdraw(sids);
Write(sids);
sids.Dispose( );
```

- В файле BankAccount.cs измените модификатор доступа конструктора **BankAccount** без параметров на **internal.**
- Сохраните проект.
- Откомпилируйте программу, исправьте ошибки, если это необходимо. Запустите программу.

  Убедитесь в том, что создается банковский счет для Sid-а и, что удачные операции по снятию и добавлению денег отражаются в списке транзакций.

➢ **Сделайте класс Bank ответственным за закрытие счетов**

Настоящие банковские счета никогда не покидают банк. Они всегда остаются внутри своего банка, а клиенты получают доступ к своим счетам, используя уникальные банковские номера. Чтобы добиться этого, Вы измените метод **Bank.CreateAccount** в файле Bank.cs.

- В класс Bank добавьте статическое **private** поле *accounts* типа **Hashtable.** Проинициализируйте его новым объектом типа **Hashtable.** Класс **Hashtable** определен внутри пространства имен **System.Collections**, поэтому необходимо добавить соответствующую директиву *using*.
- Измените метод **Bank.CreateAccount**, чтобы он возвращал номер **BankAccount** (тип **long**), а не сам **BankAccount.** В теле метода сохраните новый созданный объект **BankAccount** в хэш-таблице *accounts*, используя в качестве ключа номер банковского счета.
- В класс **Bank** добавьте нестатический **public** метод **CloseAccount**.
- Этот метод будет принимать один параметр типа **long** (номер банковского счета, который необходимо закрыть) и возвращать значение типа **bool**. В теле этого метода будет производиться доступ к объекту **BankAccount** через хэш-таблицу *accounts,* используя в качестве индекса номер банковского счета. Далее **BankAccount** будет удаляться из хэш-таблицы *accounts* через вызов метода **Remove** класса **Hashtable** и уничтожаться через вызов метода **Dispose**. Метод **CloseAccount** будет возвращать значение **true**, если удается по номеру банковского счета получить доступ к объекту **BankAccount,** в противном случае возвращается значение **false**.

  На данном этапе класс **Bank** должен выглядеть следующим образом:

```
using System.Collections;
    public class Bank
```

```
{
      public long CreateAccount( )
      {
            BankAccount newAcc = new BankAccount( );
            long accNo = newAcc.Number( );
            accounts[accNo] = newAcc;
            return accNo;
      }
      public bool CloseAccount(long accNo)
      {
            BankAccount closing =
       (BankAccount)accounts[accNo];
            if (closing != null) {
             accounts.Remove(accNo);
             closing.Dispose( );
             return true;
       }
       else {
             return false;
       }
 }
 private static Hashtable accounts = new Hashtable( );
}
```

- Сохраните проект.
- Откомпилируйте программу.

  Компиляция не пройдет. Ошибка произойдет в методе **CreateAccount.Main**, т.к. метод **Bank.CreateAccount** возвращает тип **long**, а не **BankAccount**.
- В класс **Bank** добавьте нестатический **public** метод **GetAccount**.

  Он будет принимать один параметр типа **long**, определяющий номер банковского счета. Он будет возвращать объект **BankAccount**, хранимый в хэш-таблице *accounts* с указанным номером (или **null**, если не существует банковского счета с таким номером). Объект **BankAccount** можно получить, используя номер банковского счета в качестве индекса для *accounts*:

```
public class Bank
{
      ...
      public BankAccount GetAccount(long accNo)
      {
       return (BankAccount)accounts[accNo];
      }
}
```

- В файле CreateAccount.cs измените метод **Main**, чтобы в нем использовались новые методы класса **Bank:**

```
public class CreateAccount
{
      static void Main ( )
      {
            Console.WriteLine("Sid's Account");
            Bank bank = new Bank( );
            long sidsAccNo = bank.CreateAccount( );
            BankAccount sids = bank.GetAccount(sidsAccNo);
```

```
                TestDeposit(sids);
                TestWithdraw(sids);
                Write(sids);
                if (bank.CloseAccount(sidsAccNo)) {
                 Console.WriteLine("Account closed");
                } else {
                 Console.WriteLine("Something went wrong closing
          the account");
                }
            }
            ...
    }
```

- Сохраните проект.
- Откомпилируйте программу, исправьте ошибки, если это необходимо. Запустите программу.

  Убедитесь в том, что создается банковский счет для Sid-а и, что удачные операции по снятию и добавлению денег отражаются в списке транзакций.

➢ **Для всех конструкторов BankAccount определите модификатор доступа internal**

- Найдите конструктор класса **BankAccount**, который принимает параметры типа **AccountType** и **decimal**. Поменяйте его модификатор доступа на **internal**.
- Добавьте еще один метод **CreateAccount** в класс **Bank**.

  Он будет такой же, как уже существующий метод **CreateAccount,** только с двумя параметрами типа **AccountType** и **decimal** и будет вызывать конструктор, принимающий эти два параметра.
- Найдите конструктор класса **BankAccount**, который принимает параметр типа **AccountType**. Поменяйте его модификатор доступа на **internal**.
- Добавьте третий метод **CreateAccount** в класс **Bank**.

  Он будет такой же, как два существующих метода **CreateAccount** только с одним параметром типа **AccountType** и будет вызывать конструктор, принимающий этот параметр.
- Найдите конструктор класса **BankAccount**, который принимает параметр типа **decimal**. Поменяйте его модификатор доступа на **internal**.
- Добавьте четвертый метод **CreateAccount** в класс **Bank**.

  Он будет такой же, как три существующих метода **CreateAccount** только с одним параметром типа **decimal** и будет вызывать конструктор, принимающий этот параметр.
- Сохраните проект.
- Откомпилируйте программу и исправьте ошибки, если это необходимо.

- **Запретите создание дополнительных экземпляров класса Bank**
  - Сделайте четыре перегруженных метода **Bank.CreateAccount** статическими.
  - Сделайте метод **Bank.CloseAccount** статическим.
  - Сделайте метод **Bank.GetAccount** статическим.
  - В класс **Bank** добавьте закрытый конструктор, чтобы предотвратить создание объектов класса **Bank**.
  - В файле CreateAccount.cs измените метод **CreateAccount.Main** так, чтобы в нем использовались новые статические методы и не создавался объект типа **Bank**:

```
public class CreateAccount
{
      static void Main ( )
      {
            Console.WriteLine("Sid's Account");
            long sidsAccNo = Bank.CreateAccount( );
            BankAccount sids = Bank.GetAccount(sidsAccNo);
            TestDeposit(sids);
            TestWithdraw(sids);
            Write(sids);
            if (Bank.CloseAccount(sidsAccNo))
             Console.WriteLine("Account closed");
            else
Console.WriteLine("Something went wrong closing the account");
      }
      ...
}
```

  - Сохраните проект.
  - Откомпилируйте программу, исправьте ошибки, если это необходимо. Запустите программу.

    Убедитесь в том, что создается банковский счет для Sid-а и, что удачные операции по снятию и добавлению денег отражаются в списке транзакций.

Упражнение 2. Организация классов в сборку

В этом упражнении Вы объедините классы в пространство имен **Banking,** создадите сборку и ссылку на нее. Для этого Вы сделаете следующее:

1. Поместите перечисление **AccountType** и классы **Bank, BankAccount** и **BankTransaction** в пространство имен **Banking** и откомпилируйте их в библиотеку.
2. Измените тестовое приложение. Вначале обращение к классам будет происходить по полному имени, а затем вы допишите директиву **using**.
3. Откомпилируете тестовое приложение в сборку, ссылающуюся на библиотеку **Banking.**
4. Используя ILDASM убедитесь в том, что тестовый файл ссылается на DLL Banking и не содержит сами классы **Bank** и **BankAccount**.

- **Поместите все классы в пространство имен Banking**
  - Откройте проект Bank.sln из папки *install folder*\Labs\Lab11\Exercise2\Starter\Bank.
  - Измените перечисление **AccountType** в файле AccountType.cs таким образом, чтобы оно принадлежало пространству имен **Banking:**

```
namespace Banking
{
 public enum AccountType { ... }
}
```

  - Измените класс **Bank** в файле Bank.cs таким образом, чтобы он принадлежал пространству имен **Banking:**

```
namespace Banking
{
      public class Bank
      {
            ...
      }
}
```

  - Измените класс **BankAccount** в файле BankAccount.cs таким образом, чтобы он принадлежал пространству имен **Banking:**

```
namespace Banking
{
      sealed public class BankAccount
      {
            ...
      }
}
```

  - Измените класс **BankTransaction** в файле BankTransaction.cs таким образом, чтобы он принадлежал пространству имен **Banking:**

```
namespace Banking
{
      public class BankTransaction
      {
            ...
      }
}
```

  - Сохраните проект.
  - Откомпилируйте программу. Компиляция не пройдет. В файле CreateAccount.cs ссылки на **Bank, BankAccount** и **BankTransaction** не работают, т.к. эти классы теперь принадлежат пространству имен **Banking.** Внесите изменения в **CreateAccount.Main**, указав полные имена всех этих ссылок. Например:

```
static void write(BankAccount acc) { ... }
```

    превратится в:

```
static void write(Banking.BankAccount acc) { ... }
```

  - Сохраните проект.
  - Откомпилируйте программу и исправьте ошибки, если это необходимо. Убедитесь в том, что создается банковский счет для

Sid-a и, что удачные операции по снятию и добавлению денег отражаются в списке транзакций.

- Откройте окно командной строки Visual Studio .NET.
- Из командной строки запустите ILDASM.
- Откройте Bank.exe, используя ILDASM. Файл находится в папке *install folder*\Labs\Lab11\Exercise2\Starter\Bank\bin\debug.
- Обратите внимание на то, что три класса и перечисление теперь находятся в пространстве имен **Banking,** а также имеется класс **CreateAccount.**
- Закройте ILDASM.

➢ **Создайте и воспользуйтесь библиотекой Banking**

- Откройте окно командной строки Visual Studio .NET и перейдите в папку *install folder*\Labs\Lab11\Exercise2\Starter\Bank. Из командной строки создайте банковскую библиотеку следующим образом:

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> dir
...
bank.dll
...
```

- Из командной строки запустите ILDASM, передав имя DLL в качестве параметра командной строки:

```
c:\> ildasm bank.dll
```

- Обратите внимание на то, что три “Bank*” класса и перечисление так и находятся в пространстве имен **Banking,** а класса **CreateAccount** там уже нет.
- Закройте ILDASM.
- Из командной строки откомпилируйте тестовое приложение из файла CreateAccount.cs в сборку, которая ссылается на библиотеку bank.dll:

```
c:\> csc /reference:bank.dll createaccount.cs
c:\> dir
...
createaccount.exe
...
```

- Из командной строки запустите ILDASM, передав имя exe-файла в качестве параметра командной строки:

```
c:\> ildasm createaccount.exe
```

- Обратите внимание на то, что три класса и перечисление уже не являются частью createaccount.exe. В ILDASM дважды щелкните по элементу MANIFEST, чтобы открылось окно **Manifest.** Просмотрите манифест. Обратите внимание на то, что исполняемый файл ссылается, но не содержит банковскую библиотеку:

```
.assembly extern bank
```

- Закройте ILDASM.

➢ **Упростите тестовый файл, используя директиву using**

- Отредактируйте файл CreateAccount.cs, удалив все случаи явного указания пространства имен **Banking**. Например:
  ```
  static void write(Banking.BankAccount acc) { ... }
  ```
  превратится в:
  ```
  static void write(BankAccount acc) { ... }
  ```
- Сохраните проект.
- Попытайтесь откомпилировать проект. Компиляция не пройдет. **Bank, BankAccount** и **BankTransaction** так и не удается обнаружить.
- В начале файла CreateAccount.cs добавьте инструкцию *using* для пространства имен **Banking:**
  ```
  using Banking;
  ```
- Откомпилируйте программу и исправьте ошибки, если это необходимо. Убедитесь в том, что создается банковский счет для Sid-a и, что удачные операции по снятию и добавлению денег отражаются в списке транзакций.

➢ **Исследуйте internal методы**

- В файле CreateAccount.cs отредактируйте метод **Main,** добавив одну инструкцию для создания нового объекта типа **BankTransaction**:
  ```
  static void Main( )
  {
        new BankTransaction(0.0M);
        ...
  }
  ```
- Сохраните проект.
- Откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab11\Exercise2\Starter\Bank. В командной строке наберите следующую строку, чтобы убедиться в том, что Вы можете создать исполняемый файл, который *не использует* банковскую библиотеку:
  ```
  c:\> csc /out:createaccount.exe *.cs
  ```
- Из командной строки убедитесь в том, что Вы можете создать исполняемый файл, который *использует* банковскую библиотеку:
  ```
  c:\> csc /target:library /out:bank.dll a*.cs b*.cs
  c:\> csc /reference:bank.dll createaccount.cs
  ```
- Добавленная в **Main** инструкция не создаст проблем в любом случае, т.к. конструктор **BankTransaction** в файле BankTransaction.cs объявлен как **public**.
- Отредактируйте класс **BankTransaction** в файле BankTransaction.cs, поменяв модификатор доступа конструктора на **internal**.
- Сохраните проект.
- Из командной строки убедитесь в том, что Вы *все еще* можете создать исполняемый файл, который *не использует* банковскую библиотеку:
  ```
  c:\> csc /out:createaccount.exe *.cs
  ```

- Из командной строки убедитесь в том, что Вы не можете создать исполняемый файл, который *использует* банковскую библиотеку:

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> csc /reference:bank.dll createaccount.cs
....error CS0122:
'Banking.BankTransaction.BankTransaction(decimal)' is
inaccessible due to its protection level
```

- Удалите из метода **CreateAccount.Main** инструкцию для создания нового объекта типа **BankTransaction.**
- Сохраните проект.
- Убедитесь еще раз в том, что Вы можете откомпилировать тестовый файл в сборку, ссылающуюся на библиотеку bank.dll:

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> csc /reference:bank.dll createaccount.cs
```

Лабораторная работа 11. Перегрузка операторов и использование событий

Цель работы

Изучение событийного управления при создании программ и приобретение навыков работы по перегрузке операторов.

Упражнение 1. Перегрузка операторов для класса BankAccount.

В предыдущих лабораторных работах Вы создали различные классы для обслуживания банковской системы. В классе **BankAccount** содержится подробная информация о счете клиента, включая информацию о номере и балансе счета. Вы также создали класс **Bank**, который является фабрикой для создания и управления объектами типа **BankAccount.**

В этом задании Вы переопределите операторы = = и != для класса **BankAccount.** По умолчанию эти операторы имеют реализацию, унаследованную от класса **Object**, и проверяют равенство ссылок. Вы переопределите их для проверки и сравнения информации о двух счетах.

Далее Вы переопределите методы **Equals** и **ToString.** Метод **Equals** используется средой выполнения и должен работать также как и операторы сравнения. Многие классы .NET Framework используют метод **ToString** для строкового представления объекта.

➢ **Переопределите операторы = = и !=**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab12\Starter\Bank.
- В класс **BankAccount** добавьте следующий метод:

```
public static bool operator == (BankAccount acc1,
BankAccount acc2)
{
 ...
}
```

- В тело **operator==** добавьте инструкции для сравнения двух объектов **BankAccount**. Если совпадают номера, типы и балансы счетов, то возвращается значение **true**, иначе **false**.
- Откомпилируйте проект. Вы получите сообщение об ошибке.
- В класс **BankAccount** добавьте следующий метод:

```
    public static bool operator != (BankAccount acc1,
BankAccount acc2)
    {
     ...
    }
```

- В тело **operator!=** добавьте инструкции для сравнения двух объектов **BankAccount**. Если совпадают номера, типы и балансы счетов, то возвращается значение **false**, иначе **true**. Вы можете обратить результат вызова **operator==.**
- Сохраните и откомпилируйте проект. На этот раз компиляция должна пройти успешно. Предыдущая ошибка была вызвана отсутствием парного метода для **operator==.** (Если Вы определяете оператор **operator==**, то Вы должны определить и оператор **operator!=** и наоборот).

  Итоговый код для обоих операторов должен выглядеть следующим образом:

```
public class BankAccount
{
     ...
     public static bool operator == (BankAccount acc1,
 BankAccount acc2)
     {
          if ((acc1.accNo == acc2.accNo) &&
           (acc1.accType == acc2.accType) &&
           (acc1.accBal == acc2.accBal)) {
           return true;
      } else {
           return false;
      }
     }
     public static bool operator != (BankAccount acc1,
 BankAccount acc2)
     {
      return !(acc1 == acc2);
     }
...}
```

➢ **Протестируйте операторы**

- Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab12\Starter\TestHarness.
- Создайте ссылку на компонент **Bank**, созданный в предыдущих лабораторных работах. Для этого:
  - Разверните проект TestHarness в Solution Explorer.
  - ПКМ **References** →**Add Reference**.

- Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab12\Starter\Bank\bin\debug.
- Выделите **Bank.dll**, а затем нажмите на кнопку **Open**.
- Нажмите **OK.**

- В методе **Main** класса **CreateAccount** создайте два объекта **BankAccount**. Для этого:
  - Используя **Bank.CreateAccount(),** создайте два объекта **BankAccount** с одинаковыми типами и балансами счетов.
  - Сохраните сгенерированные номера счетов в переменных *accNo1* и *accNo2*.
- Создайте две переменных *acc1* и *acc2* типа **BankAccount.** Заполните их значениями двух счетов, созданных на предыдущем шаге, вызвав метод **Bank.GetAccount()**.
- Сравните *acc1* и *acc2*, используя оператор = =. В результате должно вернуться значение **false**, т.к. номера счетов различны.
- Сравните *acc1* и *acc2*, используя оператор !=. В результате должно вернуться значение **true.**
- Создайте третью переменную *acc3* типа **BankAccount.** Заполните ее значением счета, которое вы использовали для заполнения переменной *acc1*, вызвав метод **Bank.GetAccount()** с параметром *accNo1*.
- Сравните *acc1* и *acc2*, используя оператор = =. В результате должно вернуться значение **true** т.к. счета содержат одинаковые данные.
- Сравните *acc1* и *acc2*, используя оператор !=. В результате должно вернуться значение **false**.

Если у Вас возникли какие-то проблемы, то для отображения содержимого объектов **BankAccount,** передаваемых в качестве параметров, вы можете воспользоваться функцией **Write.**

Итоговый текст кода тестирования должен выглядеть следующим образом:

```
class CreateAccount
{
      static void Main( )
      {
long accNo1 = Bank.CreateAccount(AccountType.Checking,
            100);
long accNo2 = Bank.CreateAccount(AccountType.Checking,
            100);

            BankAccount acc1 = Bank.GetAccount(accNo1);
            BankAccount acc2 = Bank.GetAccount(accNo2);

            if (acc1 == acc2) {
             Console.WriteLine("Both accounts are the same.
             They should not be!");
            } else {
             Console.WriteLine("The accounts are different.
             Good!");
```

```
            }

            if (acc1 != acc2) {
                Console.WriteLine("The accounts are
             different. Good!");
            } else {
                Console.WriteLine("Both accounts are the
             same. They should not be!");
            }

            BankAccount acc3 = Bank.GetAccount(accNo1);
            if (acc1 == acc3) {
             Console.WriteLine("The accounts are the same.
        Good!");
            } else {
                Console.WriteLine("The accounts are
             different. They should not be!");
            }

            if (acc1 != acc3) {
                Console.WriteLine("The accounts are
             different. They should not be!");
            } else {
             Console.WriteLine("The accounts are the same.
         Good!");
            }
        }
    }
```

- Откомпилируйте и запустите программу.

➢ **Переопределите методы Equals, ToString и GetHashCode**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab12\Starter\Bank.
- В класс **BankAccount** добавьте метод **Equals**:

```
public override bool Equals(object acc1)
{
...
}
```

Метод **Equals** должен иметь ту же функциональность, что и оператор = =, за исключением того, что он является не статическим методом, а методом экземпляра. Для сравнения **this** и *acc1* используйте оператор = =.

- Добавьте метод **ToString** следующим образом:

```
public override string ToString( )
{...}
```

В теле метода **ToString** должно возвращаться строковое представление экземпляра объекта.

- Добавьте метод **GetHashCode** следующим образом:

```
public override int GetHashCode( )
{...}
```

Метод **GetHashCode** должен возвращать уникальное значение для каждого уникального счета, но различные ссылки на один и тот

же счет должны возвращать одно и то же значение. Самый простой способ реализовать эту возможность через возврат номера счета. (Предварительно необходимо привести его к типу **int**).

- Итоговый код методов **Equals, ToString** и **GetHashCode** должен выглядеть следующим образом:

```
public override bool Equals(Object acc1)
{
 return this == (BankAccount)acc1;
}

public override string ToString( )
{
     string retVal = "Number: " + this.accNo + "\tType: ";
     retVal += (this.accType == AccountType.Checking) ?
 "Checking" : "Deposit";
     retVal += "\tBalance: " + this.accBal;
     return retVal;
}

public override int GetHashCode( )
{
 return (int)this.accNo;
}
```

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

➢ **Протестируйте методы Equals и ToString**

- Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab12\Starter\TestHarness.
- В методе **Main** класса **CreateAccount** вместо операторов = = и != используйте метод **Equals** следующим образом:

```
if (acc1.Equals(acc2)) {
...
}
if (!acc1.Equals(acc2)) {
...
}
```

- После **if**-инструкций добавьте три инструкции **WriteLine**, выводящие содержимое переменных *acc1, acc2* и *acc3*. Метод **WriteLine** использует метод **ToString** для представления аргументов в виде строк.

```
Console.WriteLine("acc1 - {0}", acc1);
Console.WriteLine("acc2 - {0}", acc2);
Console.WriteLine("acc3 - {0}", acc3);
```

- Для каждого объекта банковского счета вызовите метод **Dispose**.
- Откомпилируйте и запустите тестовую программу. Проверьте результаты.

Упражнение 2. Определение и использование событий

В этом упражнении Вы создадите класс **Audit**, задачей которого будет запись в текстовый файл информации обо всех изменениях балансов счетов. Счет должен быть уведомлен об изменениях с помощью события, опубликованного классом **BankAccount**.

Событие **Auditing** будет генерироваться методами **Deposit** и **Withdraw** класса **BankAccount**. Подписчиком на данное событие будет объект **Audit**.

Событие **Auditing** принимает в качестве параметра объект **BankTransaction**, который создается при каждом добавлении или снятии денег со счета.

➢ **Создайте класс, передаваемый в качестве параметра**

В этом упражнении генерируемому событию будет передаваться в качестве параметра объект **BankTransaction**. Классы, передаваемые в качестве параметров, должны наследоваться от класса **System.EventArgs**, поэтому будет создан новый класс, содержащий **BankTransaction.**

- Откройте проект Audit.sln из папки *install folder*\Labs\Lab12\Starter\Audit.
- Создайте новый класс, используя пункт меню **Project →Add New Item**. Убедитесь в том, что Вы создаете **New C# Class** и назовите его **AuditEventArgs.cs**.
- Переименуйте пространство имен в **Banking**.
- Измените определение класса **AuditEventArgs**, чтобы он наследовался от **System.EventArgs**:

```
public class AuditEventArgs : System.EventArgs
{...}
```

- Создайте закрытую переменную только для чтения типа **BankTransaction** с именем *transData* и присвойте ей значение **null**:

```
private readonly BankTransaction transData = null;
```

- Измените конструктор по умолчанию таким образом, чтобы он принимал один параметр *transaction* типа **BankTransaction** и присвойте его **this.transData**. Код конструктора должен выглядеть следующим образом:

```
public AuditEventArgs(BankTransaction transaction)
{
 this.transData = transaction;
}
```

- Создайте **public** метод-аксессор **getTransaction**, возвращающий значение **this.transData:**

```
public BankTransaction getTransaction( )
{
 return this.transData;
}
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

- **Создайте класс Audit**
  - В проекте **Audit** создайте новый класс, используя пункт меню **Project →Add New Item.** Убедитесь в том, что Вы создаете **New C# Class** и назовите его **Audit.cs**. Этот класс будет приемником события **Auditing** и записывать информацию о транзакциях в файл на диске.
  - Переименуйте пространство имен в **Banking.**
  - Добавьте директиву **using** для пространства имен **System.IO.**
  - В класс **Audit** добавьте **private** переменную *filename* типа **string**.
  - В класс **Audit** добавьте **private** переменную *auditFile* типа **StreamWriter**.

    Замечание: StreamWriter позволяет записывать информацию в файл. В лабораторной работе 5 вы использовали **StreamReader** для чтения информации из файла. В этом задании Вы воспользуйтесь методом **AppendText** класса **StreamWriter**.

    Метод **AppendText** открывает указанный файл для добавления в него текста и записывает данные в конец файла. Для записи информации в уже открытый файл Вы будете использовать метод **WriteLine**.
  - Для класса **Audit** измените конструктор по умолчанию, чтобы он принимал один строковый параметр *fileToUse*. В теле конструктора:
    - Присвойте **this.filename** значение *fileToUse*.
    - Откройте указанный файл в режиме AppendText и сохраните дескриптор файла в переменной *auditFile*.

    Итоговый текст конструктора должен выглядеть следующим образом:

```
private string filename;
private StreamWriter auditFile;

public Audit(string fileToUse)
{
      this.filename = fileToUse;
      this.auditFile = File.AppendText(fileToUse);
}
```

  - В класс **Audit** добавьте метод, который будет использоваться для приема события **Auditing** класса **BankTransaction**. Он будет выполняться, когда объект типа **BankTransaction** будет генерировать событие. Этот метод должен быть **public void** и называться **RecordTransaction**. Он будет принимать два параметра: *sender* типа **object** и *eventData* типа **AuditEventArgs**.
  - В методе **RecordTransaction:**
    - Создайте переменную *tempTrans* типа **BankTransaction**.
    - Выполните **eventData.getTransaction** и присвойте результат *tempTrans*.
    - Если *tempTrans* не **null**, для **this.auditFile** используйте метод **WriteLine** для присоединения информации о денежной сумме

tempTrans (метод **Amount()**) и дате создания (метод **When()**) в конец файла. Не закрывайте файл.

Замечание: Параметр *sender* не используется данным методом, но рекомендуется, чтобы все методы-обработчики событий использовали *sender* в качестве первого параметра.

Итоговый код должен выглядеть следующим образом:

```
public void RecordTransaction(object sender,
AuditEventArgs eventData)
{
 BankTransaction tempTrans = eventData.getTransaction( );
      if (tempTrans != null)
      this.auditFile.WriteLine("Amount: {0}\tDate: {1}",
      tempTrans.Amount( ), tempTrans.When( ));
}
```

- В класс **Audit** добавьте **private** переменную *closed* типа **bool** и присвойте ей значение **false**.
- В классе **Audit** создайте **public void** метод **Close** для закрытия **this.auditFile**. Код метода **Close** должен выглядеть следующим образом:

```
public void Close()
{
      if (!closed)
      {
       this.auditFile.Close();
       closed = true;
      }
}
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Протестируйте класс Audit**

- Откройте проект AuditTestHarness.sln из папки *install folder*\Labs\Lab12\Starter\AuditTestHarness.
- Для добавления ссылки на библиотеку, содержащую откомпилированный класс **Audit** выполните следующие действия:
  - Разверните проект AuditTestHarness в Solution Explorer.
  - ПКМ **References →Add Reference**.
  - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab12\Starter\Audit\bin\debug.
  - Выделите **Audit.dll**, а затем нажмите на кнопку **Open**.
  - Нажмите **OK.**
- Просмотрите метод **Main** класса **Test**. Этот класс:
  - Создает экземпляр класса **Audit,** используя имя AuditTrail.dat для файла, в котором будет храниться информация.
  - Создает объект типа **BankTransaction** с балансом 500 долларов.
  - Создает объект типа **AuditEventArgs**, который использует объект типа **BankTransaction.**
  - Для объекта типа **Audit** вызывается метод **RecordTransaction**.

Тест повторяется для еще одной транзакции: -200 долларов.

После второго теста вызывается метод **Close**.

- Откомпилируйте проект
- Откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab12\Starter\AuditTestHarness. В этой папке содержатся файлы AuditTestHarness.exe и Audit.dll.
- Запустите **AuditTestHarness.**
- В текстовом редакторе просмотрите содержимое файла AuditTrail.dat. В нем должна содержаться информация о двух транзакциях.

➢ **Создайте событие Auditing**

- Откройте проект Audit.sln из папки *install folder*\Labs\Lab12\Starter\Audit.
- В файле BankAccount.cs перед классом **BankAccount** объявите **public** делегат **AuditEventHandler** типа **void,** принимающий два параметра: *sender* типа **object** и *data* типа **AuditEventArgs**:

```
    public delegate void AuditEventHandler(Object sender,
AuditEventArgs data);
```

- В классе **BankAccount** объявите **private** событие **AuditTransaction** типа **AuditEventHandler** и присвойте ему значение **null:**

```
private event AuditEventHandler AuditingTransaction = null;
```

- Добавьте **public void** метод **AddOnAuditingTransaction**, который принимает один параметр *handler* типа **AuditEventHandler.** Этот метод должен добавлять *handler* в список делегатов, которые подписались на событие **AuditingTransaction**. Метод должен выглядеть следующим образом:

```
public void AddOnAuditingTransaction(AuditEventHandler handler)
    {
    this.AuditingTransaction += handler;
    }
```

- Добавьте еще один **public void** метод **RemoveOnAuditingTransaction,** который также принимает один параметр *handler* типа **AuditEventHandler.** Этот метод должен удалять *handler* из списка делегатов, которые подписались на событие **AuditingTransaction**. Метод должен выглядеть следующим образом:

```
    public void RemoveOnAuditingTransaction(AuditEventHandler
handler)
    {
    this.AuditingTransaction -= handler;
    }
```

- Добавьте третий метод, который будет использоваться объектом типа **BankAccount** для генерации события и оповещения подписчиков. Метод должен быть **protected void** и называться **OnAuditingTransaction**. Метод будет принимать один параметр *bankTrans* типа **BankTransaction**. Метод будет проверять событие

this.AuditingTransaction. Если оно содержит какие-либо делегаты, то будет создаваться объект *auditTrans* типа **AuditEventArgs**, который будет конструироваться с помощью *bankTrans.* Далее он будет выполнять делегаты, передавая себя в качестве отправителя события и параметр *auditTrans* для передачи дополнительных данных. Код этого метода должен выглядеть следующим образом:

```
    protected void OnAuditingTransaction(BankTransaction
bankTrans)
    {
         if (this.AuditingTransaction != null) {
          AuditEventArgs auditTrans = new
     AuditEventArgs(bankTrans);
          this.AuditingTransaction(this, auditTrans);
         }
    }
```

- В метод **Withdraw** класса **BankAccount** добавьте инструкцию для вызова **OnAuditingTransaction**. Передайте ему в качестве параметра объект транзакции, создаваемый методом **Withdraw.** Эту инструкцию необходимо поместить прямо перед инструкцией **return** в конце метода. Итоговый текст метода **Withdraw** должен выглядеть следующим образом:

```
    public bool Withdraw(decimal amount)
    {
         bool sufficientFunds = accBal >= amount;
         if (sufficientFunds) {
          accBal -= amount;
          BankTransaction tran = new BankTransaction(-amount);
          tranQueue.Enqueue(tran);
          this.OnAuditingTransaction(tran);
         }
         return sufficientFunds;
    }
```

- Добавьте аналогичную инструкцию в метод **Deposit**. Итоговый текст метода **Deposit** должен выглядеть следующим образом:

```
    public decimal Deposit(decimal amount)
    {
         accBal += amount;
         BankTransaction tran = new BankTransaction(amount);
         tranQueue.Enqueue(tran);
         this.OnAuditingTransaction(tran);
         return accBal;
    }
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Подпишитесь на событие Auditing**

- Последнее, что необходимо сделать – это создать объект типа **Audit**, который будет подписчиком на событие **Auditing.** Объект типа **Audit** будет частью класса **BankAccount,** и будет создаваться при создании

нового экземпляра класса **BankAccount**, для того, чтобы для каждого счета можно было производить ревизию.

В классе **BankAccount** создайте **private** переменную *accountAudit* типа **Audit:**

```
private Audit accountAudit;
```

- В класс **BankAccount** добавьте **public void** метод **AuditTrail**. Этот метод будет создавать объект типа **Audit** и подписываться на событие **Auditing.** Он будет принимать параметр типа **string**, который будет именем файла, используемого для ревизии. Метод будет:
  - Создавать *accountAudit* используя этот параметр.
  - Создавать переменную *doAuditing* типа **AuditEventHandler** и инициализировать ее, используя метод **RecordTransaction** для *accountAudit*.
  - Добавлять *doAuditing* в список подписчиков на событие **Auditing.** Использовать метод **AddOnAuditingTransaction**, передавая в качестве параметра *doAuditing*.

Итоговый код должен выглядеть следующим образом:

```
public void AuditTrail(string auditFileName)
{
     this.accountAudit = new Audit(auditFileName);
     AuditEventHandler doAuditing = new
 AuditEventHandler(this.accountAudit.RecordTransaction);
     this.AddOnAuditingTransaction(doAuditing);
}
```

- В деструкторе класса **BankAccount** добавьте инструкцию для вызова метода **Dispose** (чтобы убедиться, что все записи для ревизии корректно записаны на диск).
- В методе **Dispose** класса **BankAccount** добавьте следующую строку кода внутри инструкции **if**:

```
accountAudit.Close();
```

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Протестируйте событие Auditing**

- Откройте проект EventTestHarness.sln из папки *install folder*\Labs\Lab12\Starter\EventTestHarness.
- Для добавления ссылки на библиотеку, содержащую откомпилированные классы **Audit** и **BankAccount** выполните следующие действия:
  - Разверните проект EventTestHarness в Solution Explorer.
  - ПКМ **References →Add Reference**.
  - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab12\Starter\Audit\bin\debug.
  - Выделите **Audit.dll**, а затем нажмите на кнопку **Open**.
  - Нажмите **OK.**
- В классе **Test** просмотрите метод **Main**. Этот класс:

- Создает два банковских счета.
- Использует метод **AuditTrail** для создания для каждого из счетов встроенного объекта типа **Audit** и подписки на событие **Auditing**.
- Для каждого из счетов проводит несколько операций снятия и добавления денег.
- Закрывает оба счета.

- Откомпилируйте проект и исправьте ошибки, если это необходимо.
- Откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab12\Starter\EventTestHarness\bin\Debug. В этой папке содержаться файлы EventTestHarness.exe и Audit.dll.
- Запустите EventTestHarness
- В любом текстовом редакторе просмотрите содержимое файлов Account1.dat и Account2.dat. Они должны содержать информацию о транзакциях, проводимых для каждого из счетов.

Лабораторная работа 12. Использование свойств и индексаторов

Цель работы

Изучение свойств и индексаторов при создании классов и приобретение навыков работы с ними.

Упражнение 1. Изменение класса BankAccount

В этом упражнении Вы удалите из класса **BankAccount** методы для работы с типом и номером банковского счета и замените их свойствами только для чтения. Также Вы добавите в класс **BankAccount** свойство имя владельца счета.

➢ **Замените методы Number и Type свойствами только для чтения**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab13\Exercise1\Starter\Bank.
- В классе **BankAccount** замените метод **Number** свойством только для чтения (свойство реализует только **get**-аксессор):

```
public long Number
{
 get { return accNo; }
}
```

- Откомпилируйте проект.

  Вы получите сообщение об ошибке, т.к. **BankAccount.Number** все еще используется как метод в четырех перегруженных методах **Bank.CreateAccount**.
- Измените эти четыре метода **Bank.CreateAccount**, используя доступ к номеру счета через свойство.

  Например, замените

```
long accNo = newAcc.Number( );
```

  на:

```
long accNo = newAcc.Number;
```

- Сохраните и откомпилируйте проект.
- В классе **BankAccount** замените метод **Type** свойством только для чтения, **get**-аксессор которого возвращает **accType.ToString.**
- Сохраните и откомпилируйте проект.

➢ **Добавьте в класс BankAccount свойство для чтения-записи имя владельца счета**

- В класс **BankAccount** добавьте **private** поле *holder* типа **string**.
- В класс **BankAccount** добавьте **public** свойство **Holder** для чтения-записи типа **string**.
- **get** и **set-**аксессоры данного свойства будут использовать только что созданное вами поле **holder**:

```
public string Holder
{
 get { return holder; }
 set { holder = value; }
}
```

- Измените метод **BankAccount.ToString** таким образом, чтобы в возвращаемой им строке помимо номера, типа и баланса счета возвращалось еще и имя владельца счета.
- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

➢ **Протестируйте свойства**

- Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab13\Exercise1\Starter\TestHarness.
- Создайте ссылку на компонент **Bank**, созданный в предыдущих лабораторных работах. Для этого:
  - Разверните проект TestHarness в Solution Explorer.
  - ПКМ **References →Add Reference**.
  - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab13\Exercise1\Starter\Bank\bin\debug.
  - Выделите **Bank.dll**, а затем нажмите на кнопку **Open**.
  - Нажмите **OK.**
- В метод **Main** класса **CreateAccount** добавьте две следующих инструкции:
  - Для переменной *acc1* задайте имя владельца **“**Sid**”.**
  - Для переменной *acc2* задайте имя владельца **“**Ted**”.**
- Добавьте инструкции для извлечения и вывода на экран номера и типа каждого счета.
- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.
- Запустите проект и убедитесь в том, что на экране появятся номера, типы и имена (“Sid” и “Ted”) владельцев счетов.

Упражнение 2. Изменение класса BankTransaction

В этом упражнении Вы измените класс **BankTransaction**, разработанный в предыдущих лабораторных работах. В классе **BankTransaction** содержится информация обо всех транзакциях, производимых с объектом **BankAccount**.

Вы замените методы **When** и **Amount** парой свойств только для чтения. Метод **When** возвращает дату и время транзакции, метод **Amount** возвращает сумму, задействованную в транзакции.

- **Замените метод When свойством только для чтения**
  - Откройте проект Bank.sln из папки *install folder*\Labs\Lab13\Exercise2\Starter\Bank.
  - В классе **BankTransaction** замените метод **When** свойством только для чтения с тем же именем.
  - Откомпилируйте проект.

    Вы получите сообщение об ошибке, т.к. **BankTransaction.When** все еще используется как метод в **Audit.RecordTransaction**.
  - Измените метод **Audit.RecordTransaction**, чтобы **When** использовался как свойство.
  - Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

- **Замените метод Amount свойством только для чтения**
  - В классе **BankTransaction** замените метод **Amount** свойством только для чтения с тем же именем.
  - Откомпилируйте проект.

    Вы получите сообщение об ошибке, т.к. **BankTransaction.Amount** все еще используется как метод в **Audit.RecordTransaction**.
  - Измените метод **Audit.RecordTransaction**, чтобы **Amount** использовался как свойство.
  - Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

- **Протестируйте свойства**
  - Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab13\Exercise2\Starter\TestHarness.
  - Создайте ссылку на компонент **Bank**, созданный в предыдущих лабораторных работах. Для этого:
    - Разверните проект TestHarness в Solution Explorer.
    - ПКМ **References** →**Add Reference**.
    - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab13\Exercise2\Starter\Bank\bin\debug.
    - Выделите **Bank.dll**, а затем нажмите на кнопку **Open**.

- Нажмите **OK.**

- В метод **Main** класса **CreateAccount** добавьте следующие инструкции:
  - Добавьте деньги на счета *acc1* и *acc2*. (Используйте метод **Deposit**).
  - Снимите деньги со счетов *acc1* и *acc2*. (Используйте метод **Withdraw**).
- Для каждого счета выведите информацию обо всех, производимых с ним транзакциях. В конце класса определен метод **Write**, в который необходимый передать в качестве параметра счет, информацию о котором вы хотите просмотреть. Он использует и тестирует свойства **When** и **Amount** класса **BankTransaction.**

  Например:

```
Write(acc1);
```

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.
- Запустите проект и убедитесь в том, что информация о транзакциях отображается корректным образом.

Упражнение 3. Создание и использование индексаторов

В этом упражнении Вы добавите индексатор для класса **BankAccount** для осуществления доступа к любому из объектов **BankTransaction**, кэшируемых во внутреннем массиве.

Транзакции, производимые со счетом, доступны через очередь (**System.Collection.Queue**), которая содержится в объекте **BankAccount.**

Вы создадите индексатор для класса **BankAccount**, который будет извлекать транзакции в определенной точке очереди или возвращать значение **null**, если в указанной точке транзакций нет. Например:

```
myAcc.AccountTransactions[2]
```

возвратит транзакцию номер 2 (третью в очереди).

В этом задании вы воспользуетесь методом **GetEnumerator** класса **System.Collection.Queue.**

➢ **Для класса BankAccount объявите индексатор только для чтения**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab13\Exercise3\Starter\Bank.
- В классе **BankAccount** объявите **public** индексатор, возвращающий значения типа **BankTransaction** и принимающий в один параметр **index** типа **int**:

```
public BankTransaction this[int index]
{
...
}
```

- В тело индексатора добавьте **get**-аксессор с одной инструкцией:

```
return new BankTransaction(99);
```

Код индексатора должен выглядеть следующим образом:

```
public BankTransaction this[int index]
{
 get { return new BankTransaction(99); }
}
```

На данном шаге необходимо только протестировать синтаксис индексатора. Позже, Вы определите индексатор нужным образом.

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

➢ **Создайте транзакции**

- Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab13\Exercise3\Starter\TestHarness.
- Создайте ссылку на компонент **Bank**, созданный в предыдущих лабораторных работах. Для этого:
  - Разверните проект TestHarness в Solution Explorer.
  - ПКМ **References →Add Reference**.
  - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab13\Exercise3\Starter\Bank\bin\debug.
  - Выделите **Bank.dll**, а затем нажмите на кнопку **Open**.
  - Нажмите **OK.**
- Создайте несколько транзакций, добавив следующие инструкции в конец метод **CreateAccount.Main:**

```
for (int i = 0; i < 5; i++) {
 acc1.Deposit(100);
 acc1.Withdraw(50);
}
Write(acc1);
```

Транзакции создадутся при вызове методов **Deposit** и **Withdraw.**

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.

Запустите проект и убедитесь в том, что информация о транзакциях отражается корректным образом.

➢ **Вызовите индексатор BankAccount**

- В настоящий момент последние насколько инструкций метода **CreateAccount.Write** выводят информацию о транзакциях, используя цикл **foreach** следующим образом:

```
Queue tranQueue = acc.Transactions( );
foreach (BankTransaction tran in tranQueue) {
 Console.WriteLine("Date:  {0}\tAmount:  {1}",  tran.When,
tran.Amount);
}
```

- Выведите информацию о транзакциях другим способом:
  - Замените цикл **foreach** циклом **for,** в котором будет инкрементироваться значение переменной *counter* типа **int** в диапазоне от нуля до значения, возвращаемого **tranQueue.Count.**

- В теле цикла **for** вызовите индексатор **BankAccount,** объявленный ранее. В качестве индекса используйте *counter* и сохраняйте возвращаемое значение типа **BankAccount** в локальной переменной *tran*.
- Выведите на экран информацию из переменной *tran*:

```
    for (int counter = 0; counter < tranQueue.Count;
counter++) {
   BankTransaction tran = acc[counter];
   Console.WriteLine("Date: {0}\tAmount: {1}", tran.When,
tran.Amount);
    }
```

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.
- Запустите проект.

  На экран выведется информация о нескольких транзакциях со значением 99 (это временное тестовое значение), так как индексатор пока еще определен не полностью.

➢ **Завершите определение индексатора BankAccount**

- Вернитесь в проект Bank.sln из папки *install folder*\Labs\Lab13\Exercise3\Starter\Bank.
- В классе **BankAccount** удалите из тела индексатора инструкцию

```
return new BankTransaction(99);
```

- Транзакции **BankAccount** хранятся в private поле *tranQueue* типа **System.Collection.Queue**. У класса **Queue** нет индексатора, поэтому для доступа к нужному элементу, необходимо вручную производить итерации. Реализуется это следующим образом:
  - Объявите переменную типа **IEnumerator** и проинициализируйте ее, используя метод **GetEnumerator** для переменной *tranQueue*.
  - Производите итерации по очереди *n* раз, используя метод **MoveNext** для переменной **IEnumerator** для перехода к следующей единице очереди.
  - Верните значение **BankTransaction,** вычисленное для *n*-ой позиции.

    Ваш код должен выглядеть следующим образом:

```
IEnumerator ie = tranQueue.GetEnumerator( );
for (int i = 0; i <= index; i++) {
ie.MoveNext( );
}
BankTransaction tran = (BankTransaction)ie.Current;
return tran;
```

- Проверьте, чтобы значение параметра *index* типа **int** находилось в пределах от нуля до **tranQueue.Count.**

  Сделайте эту проверку перед проведением итераций **tranQueue.**
- Итоговый код индексатора должен выглядеть следующим образом:

```
public BankTransaction this[int index]
{
     get
```

```
        {
              if (index < 0 || index >= tranQueue.Count)
               return null;

              IEnumerator ie = tranQueue.GetEnumerator( );
              for (int i = 0; i <= index; i++) {
               ie.MoveNext( );
              }
              BankTransaction tran =
         (BankTransaction)ie.Current;
              return tran;
        }
    }
```

- Сохраните и откомпилируйте проект. Исправьте ошибки, если это необходимо.
- Вернитесь в проект TestHarness и запустите его.

  Убедитесь в том, что все десять транзакций отобразились корректно.

Лабораторная работа 13. Создание и использование атрибутов

Цель работы

Изучение возможностей связывания декларативной информации с кодом и приобретение навыков работы с атрибутами.

Упражнение 1. Использование атрибута Conditional

В этом упражнении Вы поработаете со встроенным атрибутом **Conditional**, который применяется для условного выполнения Вашего кода.

Этот атрибут может быть полезен при отладке Ваших программ, когда Вы имеете ряд процедур, выдающих отладочную информацию.

После отладки программы при компиляции release-версии вызовы ваших отладочных процедур не создаются, хотя сами процедуры компилируются.

В этом задании в класс **BankAccount** Вы добавите метод **DumpToSreen.** Этот метод будет отражать информацию о банковском счете. Вы воспользуетесь атрибутом **Conditional** для выполнения метода в зависимости от значения параметра **DEBUG_ACCOUNT**.

➢ **Примените атрибут Conditional**

- Откройте проект Bank.sln из папки *install folder*\Labs\Lab14\Starter\Bank.
- В класс **BankAccount** добавьте **public void** метод **DumpToScreen** без параметров.

  Метод должен отображать содержимое банковского счета: номер счета, его владельца, тип и баланс. Следующий код показывает пример создания такого метода:

```
public void DumpToScreen( )
{
     Console.WriteLine("Debugging account {0}. Holder is
 {1}. Type is {2}. Balance is {3}",
     this.accNo, this.holder, this.accType, this.accBal);
}
```

- Используйте этот метод в зависимости от значения символа **DEBUG_ACCOUNT.**

  Перед методом добавьте следующий атрибут **Conditional:**

  ```
  [Conditional("DEBUG_ACCOUNT")]
  ```
- Добавьте директиву **using** для пространства имен **System.Diagnostics**.
- Откомпилируйте проект и исправьте ошибки, если это необходимо.

➢ **Протестируйте атрибут Conditional**

- Откройте проект TestHarness.sln из папки *install folder*\Labs\Lab14\Starter\TestHarness.
- Создайте ссылку на библиотеку **Bank**.
  - Разверните проект TestHarness в окне Solution Explorer.
  - ПКМ **References →Add Reference**.
  - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab14\Starter\Bank\bin\debug.
  - Выделите **Bank.dll**, а затем нажмите на кнопку **Open**.
  - Нажмите **OK.**
- Просмотрите метод **Main** класса **CreateAccount**. Обратите внимание на то, что в нем создается новый банковский счет.
- В метод **Main** добавьте следующую строку кода для вызова метода **DumpToScreen** для **myAccount**:

  ```
  myAccount.DumpToScreen( );
  ```
- Сохраните проект, откомпилируйте проект и исправьте ошибки, если это необходимо.
- Запустите тестовый проект.

  Обратите внимание на то, что ничего не происходит, т.к. метод **DumpToScreen** не был вызван.

  Из командной строки запустите утилиту ILDASM для исследования *install folder*\Labs\Lab14\Starter\Bank\bin\debug\Bank.dll.

  Вы увидите, что метод **DumpToScreen** присутствует в классе **BankAccount**.
- Дважды щелкните по **DumpToScreen** для просмотра MSIL-кода

  Вы увидите в начале метода атрибут **ConditionalAttribute**. Значит ошибка в тестовом приложении. Т.к. для метода **DumpToScreen** определен атрибут **ConditionalAttribute,** среда выполнения проигнорирует вызов этого метода, если при компиляции не определен символ **DEBUG_ACCOUNT**. Вызов метода происходит, но т.к. **DEBUG_ACCOUNT** не определен, среда выполнения тут же завершает его работу.

- Закройте ILDASM.
- Вернитесь в тестовое приложение. В файле CreateAccount.cs перед первой директивой **using** добавьте следующий код, определяющий символ **DEBUG_ACCOUNT**:

```
#define DEBUG_ACCOUNT
```

- Сохраните и откомпилируйте тестовое приложение. Исправьте ошибки, если это необходимо.
- Запустите тестовое приложение.

  Обратите внимание на то, что метод **DumpToScreen** вывел информацию о счете **myAccount**.

Упражнение 2. Создание и использование пользовательского атрибута

В этом упражнении Вы создадите пользовательский атрибут **DeveloperInfoAttribute**.Этот метод позволит сохранить имя разработчика, а также время создания класса вместе с метаданными этого класса. Для атрибута будет разрешено многократное использование, т.к. в создании класса могут принимать участие сразу несколько разработчиков.

➢ **Создание класса пользовательского атрибута**

- В Visual Studio .NET создайте новый проект Microsoft Visual C# project, используя информацию из следующей таблицы:

Элемент	**Значение**
Project Type	Visual C# Projects
Template	Class Library
Name	CustomAttribute
Location	*Install folder*\Labs\Lab14\Starter

- Измените имена класса и файла с **Class1** на **DeveloperInfoAttribute**.

  Убедитесь в том, что вы изменили и имя конструктора.
- Укажите, что класс **DeveloperInfoAttribute** наследуется от класса **System.Attribute**.

  Атрибут можно будет применять только к классам, перечислениям и структурам. Разрешается также создавать несколько экземпляров этого атрибута.
- Перед определением класса добавьте следующий атрибут **AttributeUsage**:

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Enum | AttributeTargets.Struct,
AllowMultiple=true)]
```

- Атрибут **DeveloperInfoAttribute** в качестве обязательного параметра требует имя разработчика и в качестве дополнительного строкового параметра дату создания класса. Для хранения этой информации используйте следующие **private** переменные:

```
private string developerName;
private string dateCreated;
```

- Измените конструктор таким образом, чтобы он принимал один параметр *developerName* типа **string**, а в теле конструктора присвойте значение этого параметра **this.developerName**.
- Добавьте **public string** свойство только для чтения с именем **Developer**, которое можно использовать для получения значения **developerName.** Не реализовывайте **set**-аксессор.
- Добавьте еще одно **public string** свойство с именем **Date**. В этом свойстве должен быть реализован **get-**аксессор для считывания значения **dateCreated** и **set**-аксессор для записи значения в **dateCreated**.
- Откомпилируйте класс и исправьте ошибки, если это необходимо.

Т.к. класс является частью библиотеки классов, в результате компиляции создастся DLL (CustomAttribute.dll), а не отдельная исполняемая программа. Итоговый код класса **DeveloperInfoAttribute** выглядит следующим образом:

```
namespace CustomAttribute
{
     using System;
     /// <summary>
     /// This class is a custom attribute that allows
     /// the name of the developer of a class to be stored
     /// with the metadata of that class.
     /// </summary>
     [AttributeUsage(AttributeTargets.Class |
 AttributeTargets.Enum | AttributeTargets.Struct,
 AllowMultiple=true)]
     public class DeveloperInfoAttribute: System.Attribute
     {
          private string developerName;
          private string dateCreated;

          // Constructor. Developer name is the only
          // mandatory parameter for this attribute.
       public DeveloperInfoAttribute(string developerName)
          {
           this.developerName = developerName;
          }
          public string Developer
          {
               get
               {
                return developerName;
               }
          }
          // Optional parameter
          public string Date
          {
               get
               {
                return dateCreated;
               }
               set
               {
```

```
                    dateCreated = value;
                }
            }
        }
    }
```

- **Примените пользовательский атрибут к классу**
  - Сейчас Вы воспользуетесь пользовательским атрибутом **DeveloperInfo** для записи имени разработчика класса **Rational**. Откройте проект Ratioanal.sln из папки *install folder*\Labs\Lab14\Starter\Rational.
  - Создайте ссылку на ранее созданную вами библиотеку **CustomAttribute**.
    - Разверните проект Rational в окне Solution Explorer.
    - ПКМ **References →Add Reference**.
    - Нажмите на кнопку **Browse** и перейдите в папку *install folder*\Labs\Lab14\Starter\CustomAttribute\bin\debug.
    - Выделите **CustomAttribute.dll**, а затем нажмите на кнопку **Open**.
    - Нажмите **OK**
  - К классу **Rational** добавьте атрибут **CustomAttribute.DeveloperInfo**, указав свое имя в качестве имени разработчика и текущую дату в качестве дополнительного параметра:

    ```
    [CustomAttribute.DeveloperInfo("Your Name", Date="Today")]
    ```
  - Для класса **Rational** добавьте имя еще одного разработчика.
  - Откомпилируйте проект **Rational** и исправьте ошибки, если это необходимо.
  - Откройте окно командной строки и перейдите в папку *install folder*\Labs\Lab14\Starter\Rational\bin\debug.

    В этой папке должен находиться файл Rational.exe.
  - Запустите ILDASM и откройте Rational.exe.
  - Разверните пространство имен **Rational**.
  - Разверните класс **Rational.**
  - Обратите внимание на пользовательский атрибут с переданными значениями в верхней части класса.
  - Закройте ILDASM.

- **Использование отражения для получения значений атрибутов**

Использование ILDASM – это только один из способов просмотра значений атрибутов. В C#-программах Вы также можете использовать отражение. Вернитесь в Visual Studio .NET и отредактируйте класс **TestRational** проекта **Rational**.

- В методе **Main** создайте переменную *attrInfo* типа **System.Reflection.MemberInfo:**

```
public static void Main( )
{
System.Reflection.MemberInfo attrInfo;
```

```
...
```

- Вы можете использовать объекты типа **MemberInfo** для хранения информации о членах класса. Присвойте объекту типа **MemberInfo** тип **Rational,** используя оператор **typeof**:

```
attrInfo = typeof(Rational);
```

- Атрибуты класса хранятся вместе с информацией о классе. Вы можете получить значения атрибутов, используя метод **GetCustomAttributes**. Создайте массив **attrs** для хранения переменных типа **object** и, используя метод **GetCustomAttributes** для переменной **attrInfo,** найдите все пользовательские атрибуты, используемые классом **Rational:**

```
object[ ] attrs = attrInfo.GetCustomAttributes(false);
```

- Теперь вам необходимо получить информацию об атрибутах, хранимую в массиве **attrs** и вывести ее на экран. Создайте переменную *developerAttr* типа **CustomAttribute.DeveloperInfoAttribute** и присвойте ей значение первого элемента массива **attrs**, выполнив необходимое приведение типов:

```
CustomAttribute.DeveloperInfoAttribute developerAttr;
developerAttr
=(CustomAttribute.DeveloperInfoAttribute)attrs[0];
```

- Используя **get-**аксессоры атрибута **DeveloperInfoAttribute,** получите значения **Developer** и **Date** и выведите их на экран:

```
Console.WriteLine("Developer: {0}\tDate: {1}",
developerAttr.Developer, developerAttr.Date);
```

- Повторите последние два шага для второго элемента массива.

Если Вы хотите иметь возможность получать значения для нескольких атрибутов, то можете использовать цикл.

- Откомпилируйте проект и исправьте ошибки, если это необходимо.

Итоговый код метода **Main** должен выглядеть следующим образом:

```
namespace Rational
{
using System;

// Test harness
public class TestRational
{
      public static void Main( )
      {
            System.Reflection.MemberInfo attrInfo;
            attrInfo = typeof(Rational);
            object[ ] attrs =
       attrInfo.GetCustomAttributes(false);
            CustomAttribute.DeveloperInfoAttribute
       developerAttr;
            developerAttr=(CustomAttribute.DeveloperInfoAttr
       ibute)attrs[0];
            Console.WriteLine("Developer: {0}\tDate: {1}",
       developerAttr.Developer, developerAttr.Date);
            developerAttr=(CustomAttribute.DeveloperInfoAttr
       ibute)attrs[1];
```

```
            Console.WriteLine("Developer: {0}\tDate: {1}",
          developerAttr.Developer, developerAttr.Date);
        }
}
}
```

Вариант метода **Main** с использованием цикла **foreach:**

```
public static void Main( )
{
      System.Reflection.MemberInfo attrInfo;
      attrInfo = typeof(Rational);
      object[ ] attrs =
 attrInfo.GetCustomAttributes(false);

      foreach (CustomAttribute.DeveloperInfoAttribute
 devAttr in attrs)
      {
            Console.WriteLine("Developer: {0}\tDate: {1}",
       devAttr.Developer, devAttr.Date);
      }
}
```

- При запуске программы на экране должны отразиться имена и даты, переданные для класса **Rational** через атрибут **DeveloperInfoAttribute**.

Список литературы

1. *Эндрю Троелсен*. Язык программирования C# 2010 и платформа .NET 4.0. – М.,Вильямс, 2010 г. .– 1392 с.
2. *Голощапов А*. Microsoft Visual Studio 2010 (+ CD-ROM).–БХВ-Петербург, 2011.– 544с.
3. *Ник Рендольф, Дэвид Гарднер, Майкл Минутилло, Крис Андерсон* Visual Studio 2010 для профессионалов.– М.,Диалектика, 2011. – 1184 с.
4. *Алекс Макки*. Введение в .NET 4.0 и Visual Studio 2010 для профессионалов. – Издательство: Вильямс - 2010 – 416с.
5. *Джо Майо*. Microsoft Visual Studio 2010. Самоучитель – БХВ – Петербург,-2010- 450 с.

Приложение

Таблица 1.1 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 1.2 Некоторые тэги XML

Тэг	**Назначение**
<summary>…</summary>	Обеспечивает краткое описание
<remarks>…<remarks>	Обеспечивает детальное описание, может содержать другие тэги, например параграфы
<para>… **</para>**	Позволяет добавлять параграфы в тэг описания
<list type="…">…</list>	Позволяет создавать списки
<example>…</example>	Предназначен для примеров кода в комментариях
<code>…</code>	Используется в тэге example, показывает, что текст является кодом программы в примере
<see cref="*member***"/>**	Определяет ссылку на другой элемент в комментариях
<seealso cref="*member***"/>**	Аналогичен предыдущему, только ссылки попадают в секцию See Also
<exception>…</exception>	Определяет описание исключения
<permission>…</permission>	Описывает права на доступ
<param name="*name***"> … </param>**	Для описания параметров метода
<returns>… </returns>	Описывает возвращаемое значение
<value>…</value>	Применяется для описания свойств

Таблица 1.3 Флаги компилятора командной строки

Параметр	**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

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ПРОГРАММНЫХ СИСТЕМ

Кафедра **Программных систем** входит в состав нового факультета **Инфокоммуникационные технологии**, созданного решением Ученого совета университета 17 декабря 2010 г. по предложению инициативной группы сотрудников, имеющих большой опыт в реализации инфокоммуникационных проектов федерального и регионального значения.

На кафедре ведется подготовка бакалавров и магистров по направлению **210700 «Инфокоммуникационные технологии и системы связи»:**

210700.62.10 – ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОКОММУНИКАЦИОННЫЕ СИСТЕМЫ (Бакалавр)

210700.68.10 – ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОКОММУНИКАЦИОННЫЕ СИСТЕМЫ (Магистр)

Выпускники кафедры получают фундаментальную подготовку по: математике, физике, электронике, моделированию и проектированию инфокоммуникационных систем (ИКС), информатике и программированию, теории связи и теории информации.

В рамках профессионального цикла изучаются дисциплины: архитектура ИКС, технологии программирования, ИКС в Интернете, сетевые технологии, администрирование сетей Windows и UNIX, создание программного обеспечения ИКС, Web программирование, создание клиент-серверных приложений.

Область профессиональной деятельности бакалавров и магистров включает:

- сервисно-эксплуационная в сфере современных ИКС;

- расчетно-проектная при создании и поддержке сетевых услуг и сервисов;
- экспериментально-исследовательская;
- организационно-управленческая – в сфере информационного менеджмента ИКС.

Знания выпускников востребованы:

- в технических и программных системах;
- в системах и устройствах звукового вещания, электроакустики, речевой, и мультимедийной информатики;
- в средствах и методах защиты информации;
- в методах проектирования и моделирования сложных систем;
- в вопросах передачи и распределения информации в телекоммуникационных системах и сетях;
- в методах управления телекоммуникационными сетями и системами;
- в вопросах создания программного обеспечения ИКС.

Выпускники кафедры Программных систем обладают компетенциями:

- проектировщика и разработчика структур ИКС;
- специалиста по моделированию процессов сложных систем;
- разработчика алгоритмов решения задач ИКС;
- специалиста по безопасности жизнедеятельности ИКС;
- разработчика сетевых услуг и сервисов в ИКС;
- администратора сетей: UNIX и Windows;
- разработчика клиентских и клиент-серверных приложений;
- разработчика Web – приложений;
- специалиста по информационному менеджменту;
- менеджера проектов планирования развития ИКС.

Трудоустройство выпускников:

1. ОАО «Петербургская телефонная сеть»;
2. АО «ЛЕНГИПРОТРАНС»;
3. Акционерный коммерческий Сберегательный банк Российской Федерации;
4. ОАО «РИВЦ-Пулково»;
5. СПБ ГУП «Петербургский метрополитен»;
6. ООО «СоюзБалтКомплект»;
7. ООО «ОТИС Лифт»;
8. ОАО «Новые Информационные Технологии в Авиации»;
9. ООО «Т-Системс СиАйЭс» и др.

Кафедра сегодня имеет в своем составе высококвалифицированный преподавательский состав, в том числе:

- 5 кандидатов технических наук, имеющих ученые звания профессора и доцента;

- 4 старших преподавателя;
- 6 штатных совместителей, в том числе кандидатов наук, профессиональных IT - специалистов;
- 15 Сертифицированных тренеров, имеющих Западные Сертификаты фирм: Microsoft, Oracle, Cisco, Novell.

Современная техническая база; лицензионное программное обеспечение; специализированные лаборатории, оснащенные необходимым оборудованием и ПО;качественная методическая поддержка образовательных программ; широкие Партнерские связи существенно влияют на конкурентные преимущества подготовки специалистов.

Авторитет специализаций кафедры в области компьютерных технологий подтверждается Сертификатами на право проведения обучения по методикам ведущих Западных фирм - поставщиков аппаратного и программного обеспечения.

Заслуженной популярностью пользуются специализации кафедры ПС по подготовке и переподготовке профессиональных компьютерных специалистов с выдачей **Государственного Диплома** о профессиональной переподготовке по направлениям: **"Информационные технологии (инженер-программист)"** и **"Системный инженер"**, а также Диплома о дополнительном (к высшему) образованию с присвоением квалификации: **"Разработчик профессионально-ориентированных компьютерных технологий "**. В рамках этих специализаций высокопрофессиональные преподаватели готовят компетентных компьютерных специалистов по современным в России и за рубежом операционным системам, базам данных и языкам программирования ведущих фирм: Microsoft, Cisco, IBM, Intel, Oracle, Novell и др.

Профессионализм, компетентность, опыт, и качество программ подготовки и переподготовки IT- специалистов на кафедре ПС неоднократно были удостоены **высокими наградами «Компьютерная Элита» в номинации лучший учебный центр России**.

Партнеры:

1. **Microsof**t Certified Learning Solutions;
2. **Novell** Authorized Education Center;
3. **Cisco** Networking Academy;
4. **Oracle** Academy;
5. **Sun Java** Academy и др;
6. **Prometric**;
7. **VUE**.

Мы готовим квалифицированных инженеров в области инфокоммуникационных технологий с новыми знаниями, образом мышления и способностями быстрой адаптации к современным условиям труда.

Никита Алексеевич Осипов

Разработка приложений на C#

УЧЕБНОЕ ПОСОБИЕ

В авторской редакции
Редакционно-издательский отдел НИУ ИТМО
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского национального
исследовательского университета
информационных технологий, механики
и оптики
197101, Санкт-Петербург, Кронверкский пр., 49