ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Утверждаю

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
учебной-практической работы №1

учебной практики   
*УП 05. Программирование*

*Тема: «Методы»*

г. Томск – 2020 г

РАССМОТРЕННО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБОУ СПО «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Сидиков И.Д.

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Владимировна А. В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика учебной-практической работы №1 4](#_Toc22557890)

[2. Краткие теоретические сведения 5](#_Toc22557891)

[3. Задания для выполнения 25](#_Toc22557892)

[4. Индивидуальные задания 26](#_Toc22557893)

[5. Контрольные вопросы 27](#_Toc22557894)

[6. Рекомендованная литература 28](#_Toc22557895)

[ПРИЛОЖЕНИЕ А 29](#_Toc22557896)

[ПРИЛОЖЕНИЕ Б 30](#_Toc22557897)

# ОБЩАЯ ХАРАКТЕРИСТИКА УЧЕБНОЙ-ПРАКТИЧЕСКОЙ РАБОТЫ №1

**Тема:** «Методы».

**Цель работы:** получение первоначальных навыков по работе с методами на языке программирования C#

**Проверяемые компетенции:**

ОК 1, ОК 2, ОК 4, ОК 9, ПК11.2.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы.
2. Выполните предложенное практическое задание.
3. Оформите отчет по учебной практике (шаблон представлен в приложении)
4. Отправьте отчет на проверку.

**Время выполнения заданий:** 6 часов.

**Критерии оценки:**

|  |  |  |
| --- | --- | --- |
| **№** | **Критерий** | **Баллы** |
| 1 | Отчет соответствует предложенному шаблону | 1 |
| 2 | Задание выполнено правильно | 2 |
| 3 | В коде присутствует метод | 4 |
| 4 | Составлена блок-схема к программе | 2 |
| 5 | Программа работает правильно без сбоев | 2 |
| 6 | Студент ответил на контрольные вопросы по лабораторной работе | 2 |
| 7 | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 1 |
| 8 | Своевременность выполнения задания | 1 |
| Итого | | 15 |

**Перевод в пятибалльную систему оценивания:**

«отлично» - 13 - 15 баллов

«хорошо» - 10 -12 баллов

«удовлетворительно» - 7-9

«неудовлетворительно» <7 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**Метод** — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода. Метод Main является точкой входа для каждого приложения C# и вызывается общеязыковой средой выполнения (CLR) при запуске программы.

## **Сигнатуры методов**

Методы объявляются в классе, структуре или интерфейсе путем указания уровня доступа, такого как public или private, необязательных модификаторов, таких как abstract или sealed, возвращаемого значения, имени метода и всех параметров этого метода. Все эти части вместе представляют собой сигнатуру метода.

|  |
| --- |
| Примечание!  Тип возврата метода не является частью сигнатуры метода в целях перегрузки метода. Однако он является частью сигнатуры метода при определении совместимости между делегатом и методом, на который он указывает. |

Параметры метода заключаются в скобки и разделяются запятыми. Пустые скобки указывают, что параметры методу не требуются. Этот класс содержит четыре метода:

|  |
| --- |
| С# |
| abstract class Motorcycle  {  // Anyone can call this.  public void StartEngine() {/\* Method statements here \*/ }  // Only derived classes can call this.  protected void AddGas(int gallons) { /\* Method statements here \*/ }  // Derived classes can override the base class implementation.  public virtual int Drive(int miles, int speed) { /\* Method statements here \*/ return 1; }  // Derived classes must implement this.  public abstract double GetTopSpeed();  } |

## **Доступ к методу**

Вызов метода в объекте аналогичен доступу к полю. После имени объекта добавьте точку, имя метода и круглые скобки. Аргументы перечисляются в этих скобках и разделяются запятыми. Таким образом, методы класса Motorcycle могут вызываться, как показано в следующем примере:

|  |
| --- |
| С# |
| class TestMotorcycle : Motorcycle  {  public override double GetTopSpeed()  {  return 108.4;  }  static void Main()  {    TestMotorcycle moto = new TestMotorcycle();  moto.StartEngine();  moto.AddGas(15);  moto.Drive(5, 20);  double speed = moto.GetTopSpeed();  Console.WriteLine("My top speed is {0}", speed);  }  } |

## **Параметры и аргументы метода**

Определение метода задает имена и типы всех необходимых параметров. Когда вызывающий код вызывает метод, он предоставляет конкретные значения, называемые аргументами, для каждого параметра. Аргументы должны быть совместимы с типом параметра, но имя аргумента (если есть), используемое в вызывающем коде, не обязательно должно совпадать с именем параметра, указанным в методе. Пример:

|  |
| --- |
| С# |
| public void Caller()  {  int numA = 4;  // Call with an int variable.  int productA = Square(numA);  int numB = 32;  // Call with another int variable.  int productB = Square(numB);  // Call with an integer literal.  int productC = Square(12);  // Call with an expression that evaulates to int.  productC = Square(productA \* 3);  }  int Square(int i)  {  // Store input argument in a local variable.  int input = i;  return input \* input;  } |

Параметры позволяют передать в метод некоторые входные данные. Например, определеим метод, который складывает два числа:

|  |
| --- |
| С# |
| static int Sum(int x, int y)  {      return x + y;  } |

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа.

|  |
| --- |
| С# |
| class Program  {  static void Main(string[] args)  {  int result = Sum(10, 15);  Console.WriteLine(result); // 25    Console.ReadKey();  }  static int Sum(int x, int y)  {  return x + y;  }  } |

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y. Значения, которые передаются параметрам, еще называются аргументами. То есть передаваемые числа 10 и 15 в данном случае являются аргументами.

Иногда можно встретить такие определения как формальные параметры и фактические параметры. Формальные параметры — это собственно параметры метода (в данном случае x и y), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры — это и есть аргументы метода.

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          int a = 25;          int b = 35;          int result = Sum(a, b);          Console.WriteLine(result);  // 60            result = Sum(b, 45);          Console.WriteLine(result);  // 80            result = Sum(a + b + 12, 18); // "a + b + 12" представляет значение параметра x          Console.WriteLine(result);  // 90          Console.ReadKey();      }      static int Sum(int x, int y)      {          return x + y;      }  } |

Если параметрами метода передаются значения переменных, которые представляют базовые примитивные типы (за исключением типа object), то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          int a;          int b = 9;          Sum(a, b);  // Ошибка - переменной a не присвоено значение          Console.ReadKey();      }      static int Sum(int x, int y)      {          return x + y;      }  } |

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

|  |
| --- |
| С# |
| class Program  {  static void Main(string[] args)  {  Display("Tom", 24); // Name: Tom Age: 24    Console.ReadKey();  }  static void Display(string name, int age)  {  Console.WriteLine($"Name: {name} Age: {age}");  }  } |

В данном случае первый параметр метода Display представляет тип string, поэтому мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int, поэтому должны передать ему целое число, которое соответствует типу int.

Другие данные параметрам мы передать не можем. Например, следующий вызов метода Display будет ошибочным:

|  |
| --- |
| С# |
| Display(45, "Bob"); // Ошибка! несоответствие значений типам параметров |

**Необязательные параметры**

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

|  |
| --- |
| С# |
| static int OptionalParam(int x, int y, int z=5, int s=4)  {  return x + y + z + s;  } |

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  OptionalParam(2, 3);    OptionalParam(2,3,10);    Console.ReadKey();  } |

**Именованные параметры**

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

|  |
| --- |
| С# |
| static int OptionalParam(int x, int y, int z=5, int s=4)  {  return x + y + z + s;  }  static void Main(string[] args)  {  OptionalParam(x:2, y:3);    //Необязательный параметр z использует значение по умолчанию  OptionalParam(y:2, x:3, s:10);    Console.ReadKey();  } |

## **Передача по ссылке и передача по значению**

По умолчанию при передаче в метод экземпляра типа значения вместо самого этого экземпляра передается его копия. Поэтому изменения в аргументе не оказывают влияния на исходный экземпляр в вызывающем методе. Чтобы передать экземпляр типа значения по ссылке, используйте ключевое слово ref. Дополнительные сведения см. в разделе Передача параметров типа значения.

При передаче в метод объекта ссылочного типа передается ссылка на этот объект. То есть метод получает не сам объект, а аргумент, который указывает расположение объекта. При изменении члена объекта с помощью этой ссылки это изменение отражается в аргументе в вызывающем методе, даже если объект передается по значению.

Ссылочный тип создается с помощью ключевого слова class, как показано в следующем примере.

|  |
| --- |
| С# |
| public class SampleRefType  {  public int value;  } |

Теперь, если передать объект, основанный на этом типе, в метод, то будет передана ссылка на объект. В следующем примере объект типа SampleRefType передается в метод ModifyObject:

|  |
| --- |
| С# |
| public static void TestRefType()  {  SampleRefType rt = new SampleRefType();  rt.value = 44;  ModifyObject(rt);  Console.WriteLine(rt.value);  }  static void ModifyObject(SampleRefType obj)  {  obj.value = 33;  } |

В этом примере, в сущности, делается то же, что и в предыдущем примере, — аргумент по значению передается в метод. Но поскольку здесь используется ссылочный тип, результат будет другим. В данном случае в методе ModifyObject изменено поле value параметра obj, а также изменено поле value аргумента, rtв методе TestRefType. В качестве выходных данных метод TestRefType отображает 33.

## **Возвращаемые значения**

Методы могут возвращать значение вызывающему объекту. Если тип возврата, указываемый перед именем метода, не void, этот метод может возвращать значение с помощью ключевого слова return. Инструкция с ключевым словом return , за которым следует значение, соответствующее типу возврата, будет возвращать это значение объекту, вызвавшему метод.

Значение может возвращаться вызывающему объекту по значению или, начиная с C# версии 7.0, по ссылке. Значения возвращаются вызывающему объекту по ссылке, если ключевое слово ref используется в сигнатуре метода и указывается после каждого ключевого слова return. Например, следующая сигнатура метода и оператор return указывают, что метод возвращает имена переменных estDistance вызывающему объекту по ссылке.

|  |
| --- |
| С# |
| public ref double GetEstimatedDistance()  {  return ref estDistance;  } |

Ключевое слове return также останавливает выполнение метода. Если тип возврата — void, инструкцию return без значения по-прежнему можно использовать для завершения выполнения метода. Без ключевого слова return этот метод будет останавливать выполнение при достижении конца блока кода. Методы с типом возврата, отличным от void, должны использовать ключевое слово return для возврата значения. Например, в следующих двух методах ключевое слово return используется для возврата целочисленных значений.

|  |
| --- |
| С# |
| class SimpleMath  {  public int AddTwoNumbers(int number1, int number2)  {  return number1 + number2;  }  public int SquareANumber(int number)  {  return number \* number;  }  } |

Чтобы использовать значение, возвращаемое из метода, вызывающий метод может применять сам вызов метода везде, где будет достаточно значения того же типа. Можно также назначить возвращаемое значение переменной. Например, следующие два примера кода достигают одной и той же цели.

|  |
| --- |
| С# |
| int result = obj.AddTwoNumbers(1, 2);  result = obj.SquareANumber(result);  // The result is 9.  Console.WriteLine(result); |

|  |
| --- |
| С# |
| result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));  // The result is 9.  Console.WriteLine(result); |

Использование локальной переменной, в данном случае result, для сохранения значения является необязательным. Это может улучшить читаемость кода или может оказаться необходимым, если нужно сохранить исходное значение аргумента для всей области метода.

Чтобы использовать значение, возвращаемое по ссылке из метода, необходимо объявить локальную ссылочную переменную, если планируется изменение значения. Например, если метод Planet.GetEstimatedDistance возвращает значение Double по ссылке, можно определить его как локальную ссылочную переменную с использованием кода следующего вида:

|  |
| --- |
| С# |
| ref int distance = plant |

Возвращать многомерный массив из метода M, который изменяет содержимое массива, необязательно, если вызывающая функция передает массив в M. В целях оптимизации можно возвращать полученный массив из M или функциональный поток значений, однако это необязательно. Это связано с тем, что C# передает все ссылочные типы по значению, а значение ссылки на массив представляет собой указатель на массив. В методе M любые изменения содержимого массива отслеживаются любым кодом, имеющим ссылку на массив, как показано в приведенном ниже примере:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  int[,] matrix = new int[2, 2];  FillMatrix(matrix);  // matrix is now full of -1  }  public static void FillMatrix(int[,] matrix)  {  for (int i = 0; i < matrix.GetLength(0); i++)  {  for (int j = 0; j < matrix.GetLength(1); j++)  {  matrix[i, j] = -1;  }  }  } |

## **Асинхронные методы**

С помощью функции async можно вызывать асинхронные методы, не прибегая к использованию явных обратных вызовов или ручному разделению кода между несколькими методами или лямбда-выражениями.

Если пометить метод с помощью модификатора async, можно использовать в этом методе инструкцию await . Когда управление достигает выражения await в асинхронном методе, управление возвращается вызывающему объекту и выполнение метода приостанавливается до завершения выполнения ожидающей задачи. После завершения задачи можно возобновить выполнение в методе.

|  |
| --- |
| Примечание!  Асинхронный метод возвращается в вызывающий объект, когда он встречает первый ожидаемый объект, выполнение которого еще не завершено, или когда выполнение асинхронного метода доходит до конца — в зависимости от того, что происходит раньше. |

Асинхронный метод может иметь тип возврата Task<TResult>, Taskили void. Тип возврата void в основном используется для определения обработчиков событий, где требуется тип возврата void. Асинхронный метод, который возвращает тип void, не может быть ожидающим. Вызывающий объект метода, возвращающего значение типа void, не может перехватывать исключения, которые выдает этот метод.

В следующем примере DelayAsync является асинхронным методом с типом возврата Task<TResult>. DelayAsync имеет инструкцию return , которая возвращает целое число. Поэтому объявление метода DelayAsync должно иметь тип возврата Task<int>. Поскольку тип возврата — Task<int>, вычисление выражения await в DoSomethingAsync создает целое число, как показывает следующая инструкция: int result = await delayTask.

В следующем примере метод startButton\_Click служит примером асинхронного метода с типом возврата void. Поскольку DoSomethingAsync является асинхронным методом, задача для вызова DoSomethingAsync должна быть ожидаемой, как показывает следующая инструкция: await DoSomethingAsync();. Метод startButton\_Click должен быть определен с модификатором async , так как этот метод имеет выражение await .

|  |
| --- |
| С# |
| // using System.Diagnostics;  // using System.Threading.Tasks;  // This Click event is marked with the async modifier.  private async void startButton\_Click(object sender, RoutedEventArgs e)  {  await DoSomethingAsync();  }  private async Task DoSomethingAsync()  {  Task<int> delayTask = DelayAsync();  int result = await delayTask;  // The previous two statements may be combined into  // the following statement.  //int result = await DelayAsync();  Debug.WriteLine("Result: " + result);  }  private async Task<int> DelayAsync()  {  await Task.Delay(100);  return 5;  }  // Output:  // Result: 5 |

Асинхронный метод не может объявить все параметры ref или out , но может вызывать методы, которые имеют такие параметры.

## **Определения текста выражений**

Часто используются определения методов, которые просто немедленно возвращаются с результатом выражения или которые имеют единственную инструкцию в тексте метода. Для определения таких методов существует сокращенный синтаксис с использованием =>:

|  |
| --- |
| С# |
| public Point Move(int dx, int dy) => new Point(x + dx, y + dy);  public void Print() => Console.WriteLine(First + " " + Last);  // Works with operators, properties, and indexers too.  public static Complex operator +(Complex a, Complex b) => a.Add(b);  public string Name => First + " " + Last;  public Customer this[long id] => store.LookupCustomer(id); |

Если метод возвращает void или является асинхронным методом, то текст метода должен быть выражением инструкции (так же, как при использовании лямбда-выражений). Свойства и индексаторы должны быть только для чтения, и вы не должны использовать ключевое слово get метода доступа.

## **Методы расширения**

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          string s = "Привет мир";          char c = 'и';          int i = s.CharCount(c);          Console.WriteLine(i);            Console.Read();      }  }    public static class StringExtension  {      public static int CharCount(this string str, char c)      {          int counter = 0;          for (int i = 0; i<str.Length; i++)          {              if (str[i] == c)                  counter++;          }          return counter;      }  } |

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. В данном случае это класс StringExtension. Затем объявляем статический метод. Суть нашего метода расширения - подсчет количества определенных символов в строке.

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

Затем у всех строк мы можем вызвать данный метод: int i = s.CharCount(c);. Причем нам уже не надо указывать первый параметр. Значения для остальных параметров передаются в обычном порядке.

Применение методов расширения очень удобно, но при этом надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Также следует учитывать, что методы расширения действуют на уровне пространства имен. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву using.

## **Частичные методы**

Частичные классы могут содержать частичные методы. Такие методы также опреляются с ключевым словом partial. Причем определение частичного метода без тела метода находится в одном частичном классе, а реализация этого же метода - в другом частичном классе. Например, изменим выше определенные классы Person. Первый класс:

|  |
| --- |
| С# |
| public partial class Person  {      partial void DoSomethingElse();        public void DoSomething()      {          Console.WriteLine("Start");          DoSomethingElse();          Console.WriteLine("Finish");      }  } |

Второй класс:

|  |
| --- |
| С# |
| public partial class Person  {      partial void DoSomethingElse()      {          Console.WriteLine("I am reading a book");      }  } |

В первом классе определен метод DoSomethingElse, который вызывается в методе DoSomething. Причем на момент определения первого класса неизвестно, что представляет собой метод DoSomethingElse. Тем не менее мы знаем список его параметров и может вызвать в первом классе.

Второй класс уже непосредственно определяет тело метода DoSomethingElse.

При этом частичные методы не могут иметь модификаторов доступа - по умолчанию они все считаются приватными. Также частичные методы не могут иметь таких модификаторов как virtual, abstract, override, new, sealed. Хотя допустимы статические частичные методы.

Кроме того, частичные методы не могут возвращать значения, то есть они всегда имеют тип void. И также они не могут иметь out-параметров.

Поскольку частичные методы всегда приватные, то мы не сможем иx вызвать напрямую в программе вне классов, где они определены. Поэтому обычно они вызываются через другие доступные методы как в случае выше через метод DoSomething:

|  |
| --- |
| С# |
| Person tom = new Person();  tom.DoSomething(); |

## **Перезагрузка методов**

**Перегрузка методов** – создание одноименных методов в пределах одного класса, которые отличаются количеством и/или типом параметров. Перегруженные методы могут возвращать значения разных типов данных, однако отличие только в возвращаемом типе не допускается.

**Сигнатуры**

**Сигнатура метода** – это часть объявления метода, которая позволяет компилятору идентифицировать метод среди других.

В сигнатуру входят:

* Имя метода;
* Количество параметров;
* Порядок параметров;
* Тип параметров;
* Модификаторы параметров.

Названия параметров и тип возвращаемого значения не относится к сигнатуре.

Опираясь на сигнатуру, компилятор выбирает метод, который нужно использовать.

Рассмотрим несколько методов:

|  |
| --- |
| С# |
| int Div(int a, int b)  {  return a / b;  }  uint Sum(uint x, uint y, uint z)  {  return x + y + z;  } |

Метод Div имеет следующую сигнатуру – Div(int, int), а метод Sum – Sum(uint, uint, uint).

**Перегрузка**

Исходя из понятия сигнатуры, перегруженными называют методы, которые отличаются сигнатурами, но при этом имеют одинаковые имена.

Пример перегрузки:

|  |
| --- |
| С# |
| public int Mult(int a, int b)  {  return a \* b;  }  public double Mult(double x, double y)  {  return x \* y;  }  public double Mult(double x, double y, double z)  {  //вызывает предыдущий метод  return Mult(x, y) \* z;  }  public string Mult(string s, uint k)  {  var retVal = string.Empty;  for (var i = 0; i < k; i++)  {  retVal += s;  }  return retVal;  } |

Как можно заметить, в каждом из рассмотренных примеров использована уникальная сигнатура.

C# также поддерживает сокращенную запись перегруженных методов:

|  |
| --- |
| С# |
| float F(float x) => x - 2 / x;  int F(int x) => x - 2 / x; |

При вызове метода с использованием литералов, можно указывать их тип с помощью суффиксов:

|  |
| --- |
| С# |
| var r1 = F(3); //3  var r2 = F(3f); //2.33 |

Перегруженные методы могут отличаться только модификаторами:

|  |
| --- |
| С# |
| int PlusOne(int i)  {  return i + 1;  }  int PlusOne(ref int i)  {  i++;  return i;  } |

Перегрузка используется для создания универсальных методов, логика поведения которых одинакова, но типы данных или количество аргументов разное. Это дает возможность писать красивый код, группируя методы с одинаковым поведением по имени.

Рассмотрим пример поиска минимального значения из двух целых чисел:

|  |
| --- |
| С# |
| static int Min(int n1, int n2)  {  return n1 < n2 ? n1 : n2;  } |

Используя перегрузку можно увеличить количество аргументов, для нахождения минимального из трех чисел:

|  |
| --- |
| С# |
| static int Min(int n1, int n2, int n3)  {  //вызов предыдущего метода  var m = Min(n1, n2);  return m < n3 ? m : n3;  } |

## **Ограничения при перезагрузке методов**

**Локальные функции**

C# не поддерживает перегрузку локальных функций, поэтому такой код не скомпилируется:

|  |
| --- |
| С# |
| public void MyMethod()  {  string Hello(string name) => "Hi! " + name;  string Hello(string firstName, string lastName) => "Hello! " + firstName + " " + lastName;  Console.WriteLine(Hello("John"));  Console.WriteLine(Hello("James", "Smith"));  } |

**Отличие только по возвращаемому типу**

Нельзя перегружать методы, если они отличаются только по типу возвращаемого значения. Следующий код не скомпилируется:

|  |
| --- |
| С# |
| void DisplayNumber(long l)  {  Console.Write(l);  }  long DisplayNumber(long l)  {  Console.WriteLine(l);  return l;  } |

**Методы с опциональными параметрами**

Рассмотрим метод с параметрами по умолчанию:

|  |
| --- |
| С# |
| static void ShowSum(byte a, byte b, byte c = 5)  {  Console.WriteLine(a + b + c);  } |

Может показаться, что к этому методу подходят сразу две сигнатуры: ShowSum(byte, byte, byte) и ShowSum(byte, byte), но это не так, подходит только первый вариант. Поэтому если перегрузить его методом с двумя параметрами:

|  |
| --- |
| С# |
| static void ShowSum(byte x, byte y)  {  Console.WriteLine(x + y);  } |

он будет иметь больший приоритет, и аргумент по умолчанию не используется. Выйти из ситуации можно используя именованные параметры:

|  |
| --- |
| С# |
| ShowSum(2, 3); //5, а не 10, как можно ожидать  ShowSum(2, 3, 1); //6  ShowSum(a: 2, b: 3); //10 |

Хотя при разработке программ, таких конструкций лучше избегать, потому что они вносят неоднозначность.

На практике перегрузки встречаются очень часто. К примеру, метод Console.Write(), универсальный метод, который может принимать разное количество аргументов разного типа данных, при этом поведение его во всех случаях одинакова – вывод значений на экран консольного приложения.

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Решите задание используя методы, по вариантам
2. Спроектируйте блок схему к программе
3. Отобразить в отчете проделанные шаги
4. Ответить на контрольные вопросы
5. Написать вывод о проделанной работе

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

|  |
| --- |
| Вариант 1 – Написать программу, для определения максимального из трех вещественных чисел, которые вводятся с клавиатуры, используя перегрузку метода для поиска максимального из двух значений. |
| Вариант 2 – Напишите перегруженные методы, в сокращенной форме(лямбда), которые принимают на вход два целых числа, находят их произведение и в зависимости от типа прибавляют к результату: 1 – для uint, 2 – для int, 3 – для long. |
| Вариант 3 – Написать программу для вывода n первых членов арифметической прогрессии 1, 2, 3… с использованием рекурсии и методов. |
| Вариант 4 - Напишите программу для получения суммы n первых членов арифметической прогрессии. Разность прогрессии d задается в качестве параметра. Используйте при реализации методы! |
| Вариант 5 - Напишите программу для переворота строки с использованием рекурсии и методов. |
| Вариант 6 - Напишите программу для поиска индекса максимального элемента массива с использованием рекурсии и методов. |
| Вариант 7 - Напишите программу, которая сравнивает, введенное пользователем, целое число с нулем, и выводит одно из сообщений. Используйте при реализации методы! |
| Вариант 8 –Напишите программу, которая возвращает номер самого последнего элемента из массива и который совпадает с заданным с клавиатуры числом. Используйте при реализации методы! |
| Вариант 9 –Напишите программу, которая создает двумерный массив 5 x 5. Используя метод, заполните его случайными числами от 30 до 60. Найдите максимальный и минимальный элемент массива. |
| Вариант 10 –Напишите программу, которая объявляет два целочисленных массива с разными размерами и напишите метод, который заполнит их элементами и покажет на экран. Метод доложен принимать два параметра – массив и его размер. |

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое асинхронные методы?
2. Для чего используются перегруженные методы?
3. Какие бывают ограничения на методы?

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Официальная документация Microsoft C# — Текст: электронный // Microsoft [сайт]. — URL: https://docs.microsoft.com/ (дата обращения: 12.03.2020).
2. Сообщество IT-специалистов — Текст: электронный // Habr [сайт]. — URL: https://habr.com/ (дата обращения: 12.03.2020).
3. Сайт о программирование / — Текст: электронный // Metanit [сайт]. — https://metanit.com/ (дата обращения: 12.03.2020).

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Задачи
   2. Код
   3. Блок-схема
   4. Результат работы
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по учебной-практической работе №1

учебной практике   
*УП 05. Программирование*

*Тема: «Методы»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель

Владимирова А.В.

г. Томск – 2020 г