# Методы

Определение

Методом — поименованная или иным образом идентифицированная часть [компьютерной программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы. В ООП метод принадлежит какому-то классу или объекту. © Wikipedia

**Функция** - подпрограмма, выполняющая какие-либо операции и возвращающая значение.  
**Процедура** - подпрограмма, которая только выполняет операции, без возврата значения.  
**Метод** - это функция или процедура, которая принадлежит классу или экземпляру класса.

|  |
| --- |
| [модификаторы] возвращаемый\_тип имя\_метода ([параметры])  {  //Тело цикла  } |

Самый первый метод который нас встречает в любой программе на C# это метод Main

|  |
| --- |
| static void Main(string[] args)  {  } |

Где static является модификаторам.

void возвращаемым значением. В данном слово void означает что метод ничего не возвращает.

Main именем метода.

string[] args - параметрами метода.

Модификаторы и параметры не обязательно указывать.

Пример:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  }  static void SayHelloWorld()  {  Console.WriteLine("Hello World");  }  static void DoPrograming()  {  Console.WriteLine("I Learn how to Code");  }  } |

Тут мы определили 2 метода в классе Program - SayHelloWorld() и DoPrograming().

Оба метода ничего не возвращают (возвращаемый тип void), и имеют модификатор static.

Оба метода выводят на консоль надпись. Но так как эти методы ни кто не вызывает то и запущенная программа ничего не выведет на экран.

Чтобы эти методы сработали их нужно вызвать в главном методе - методе Main:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  SayHelloWorld();  DoPrograming();  }  static void SayHelloWorld()  {  Console.WriteLine("Hello World");  }  static void DoPrograming()  {  Console.WriteLine("I Learn how to Code");  }  } |

Теперь, запустив программу, мы увидим на экране две строчки из методов. Если обобщить, то вызов метода можно привести к виду:

имя\_метода (параметры\_передаваемые\_методу);

В нашем случае методы SayHelloWorld() и DoPrograming() не имеют параметров и не указываются в скобках при вызове.

**Возвращаемое значение из метода.**

Очень часто бывает так, что метод должен не просто что-то выводить на экран, а выполнять какие-то операции и возвращать результат этих операций:

|  |
| --- |
| static double GetSum()  {  double first = 5,6;  double second = 10,1;  double result = first + second;  return result;  } |

Данный метод имеет тип возвращаемого значения double и поэтому должен возвращать именно тип double. Поэтому переменная result имеет тип double.

Для возврата значения из метода используется оператор return, после него идёт возвращаемое значение. Если после оператора return идёт значение отличающееся от указанного в методе, то будет ошибка компиляции.

Ещё пример:

|  |
| --- |
| static char FindChar()  {  int number = 4;  string str = "Hello";  char sym = str[number];  return sym;  } |

Этот метод ищет символ в строке и возвращает его значение.

Результат методов мы можем присвоить переменным:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  double sum = GetSum();  char symbol = FindChar();  Console.WriteLine(sum); // 15.7  Console.WriteLine(symbol); // o  Console.Read();  }  static double GetSum()  {  double first = 5.6;  double second = 10.1;  double result = first + second;  return result;  }  static char FindChar()  {  int number = 4;  string str = "Hello";  char sym = str[number];  return sym;  }  } |

После оператора return выполнение метода прекращается и все операции указанные после этого оператора будут игнорироваться.

Если метод не имеет возвращаемого значения, тогда оператор return выполняет выход из метода, при этом после него не должно стоять ни каких значений.

|  |
| --- |
| static void GetHello()  {  string message = "Hello";  return;  Console.WriteLine(message);  } |

Этот метод не делает ничего, зайдя в метод программа присвоит значение переменной message и сразу выйдет из метода.

Методы также могут возвращать сложные типы данных, к примеру массивы:

|  |
| --- |
| static int[] TakeCount()  {  return new int[] { 0, 1, 2, 3 };  } |

**Параметры методов**

Как правило нам необходимо чтобы метод сделал «что-то» с внешними данными которые недоступны из метода. Эти данные можно передать в виде параметров, к примеру изменим наш и методы GetSum()и FindChar():

|  |
| --- |
| static double GetSum(double first, double second)  {  return first + second;  }  static char FindChar(string str, int index)  {  return str[index];  } |

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  double sum = GetSum(12.3, 54.8);  char symbol = FindChar("Programming!", 5);  Console.WriteLine(sum); // 67.1  Console.WriteLine(symbol); // a  Console.Read();  }  static double GetSum(double first, double second)  {  return first + second;  }  static char FindChar(string str, int index)  {  return str[index];  }  } |

Теперь наши методы работают с входными данными. Значения, передаваемые параметрам называются **аргументами**, т.е. в нашем примере 12,3 и 54,8 являются аргументами, а double first и double second – параметрами. И их типы должны совпадать или должны иметь неявно приведение друг к другу.

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.

В аргументах могут быть записаны не только переменные, но и сложные выражения:

|  |
| --- |
| double sum = GetSum(12.3 \* 2 + 31 -6.5, 54.8); |

В методы можно передавать сложные типы данных, к примеру массивы:

|  |
| --- |
| static int TakeSum(int[] numbers)  {  int sum = 0;  foreach (int num in numbers)  sum += num;  return sum;  } |

**Необязательные параметры**

При вызове метода с параметрами обязательно предоставлять аргументы для всех его параметров. Но C# позволяет задать значения параметров по умолчанию:

|  |
| --- |
| static double Division(double first, double second = 2)  {  return first / second;  } |

Необязательных параметров может быть несколько:

|  |
| --- |
| static int Sum(int x, int y, int z, int a = 10, int b = 12,)  {  return x + y + z + a + b;  } |

Но необязательные параметры должны идти последними в списке и после них все параметры должны быть необязательными.

Теперь мы можем вызвать метод опустив второй параметр:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  double div = Division(12); // 6  int sum = Sum(2, 3, 5); // 32  sum = Sum(2, 3, 5, 6); // 28  Console.Read();  }  static double Division(double first, double second = 2)  {  return first / second;  }  static int Sum(int x, int y, int z, int a = 10, int b = 12,)  {  return x + y + z + a + b;  }  } |

Мы можем опустить необязательные методы, но только в порядке их следования.

**Именованные параметры**

C# позволяет указывать параметры в произвольном порядке используя именованные параметры.

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  int sum = Sum(y:10, x:5, b:20);  Console.Read();  }  static int Sum(int x, int y, int a = 10, int b = 12)  {  return x + y + a + b;  }  } |

**Передача параметров по ссылке и по значению**

Существует два способа передачи параметров по значению и по ссылке.

Передача параметров по значению фактически является обычной передачей параметров в метод:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  double div = Division(12); // 6  Console.Read();  }  static double Division(double first, double second = 2)  {  return first / second;  }  } |

Передача параметров по ссылке.

Для передачи параметров по ссылке используется ключевое слово ref:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  int x = 5;  int y = 2;  Add(ref x, y);  Console.WriteLine($"x: {x}, y: {y}"); // x: 10, y = 2  Console.Read();  }  static void Add(ref int x, int y)  {  x = x \* y;  }  } |

Для указания передачи по ссылке необходимо указать ключевое слово ref в методе перед нужным параметром и при вызове этого метода необходимо указывать ключевое слово ref.

При передаче по значению мы передаём в метод копию значения, и чтобы мы не делали в самом методе с этой переменной исходная переменная останется неизменной.

При передаче по ссылке мы фактически передаем ссылку на адрес в памяти и изменяя значение переменной в методе, значение исходной переменной тоже измениться.

**Модификатор out**

Параметры могут быть не только входными, но выходными. Для того чтобы обозначить выходной параметр необходимо перед ним указать ключевое слово out.

|  |
| --- |
| static void Sum(int x, int y, out int a)  {  a = x + y;  } |

Как и в случае с ref, ключевое слово out необходимо указывать как при определении метода так и при его вызове.

При использовании выходного параметра, в методе обязательно присвоить значение этому параметру.

Основное преимущества использования выходных параметров это возможность вернуть из метода несколько значений:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  int x = 10;  int area;  int perimetr;  GetData(x, 15, out area, out perimetr);  Console.WriteLine("Площадь : " + area);  Console.WriteLine("Периметр : " + perimetr);  Console.ReadKey();  }  static void GetData(int x, int y, out int area, out int perim)  {  area = x \* y;  perim = (x + y) \* 2;  }  } |

В этом примере мы получаем из метода сразу два значения: площадь и периметр.

Так как значения переменных передаваемых в метод с ключевым словом out полностью обнуляются и мы не можем их использовать в методе, то допустима следующая запись:

|  |
| --- |
| int x = 10;  GetData(x, 15, out int area, out int perimetr);  Console.WriteLine($"Площадь : {area}");  Console.WriteLine($"Периметр : {perimetr}"); |

В этом примере мы объявляем переменные при вызове метода и в дальнейшем они будут доступны в вызывающем методе.

**Массив параметров, ключевое слово params**

С помощью ключевого слова params мы можем передавать в методы неопределённое количество параметров:

|  |
| --- |
| static void Main(string[] args)  {  int sum = TakeSum(1, 2, 3, 4, 5, 25);  Console.ReadKey();  }  static int TakeSum(params int[] numbers)  {  int sum = 0;  foreach (int num in numbers)  sum += num;  return sum;  } |

Как видно из примера сам метод слабо отличается от предыдущего, где мы просто передавали массив в качестве параметра, но их вызов отличается достаточно сильно:

|  |
| --- |
| static void Main(string[] args)  {  int[] numbers = new int[] { 1, 2, 3, 4, 5, 25 };  int sumArray = TakeSumArray(numbers);  int sumParams = TakeSumParams(1, 2, 3, 4, 5, 25);  sumParams = TakeSumParams(1);  sumParams = TakeSumParams();  sumParams = TakeSumParams(numbers);  Console.ReadKey();  }  static int TakeSumParams(params int[] numbers)  {  int sum = 0;  foreach (int num in numbers)  sum += num;  return sum;  }  static int TakeSumArray(int[] numbers)  {  int sum = 0;  foreach (int num in numbers)  sum += num;  return sum;  } |

В примере видно, что если мы вызываем метод у которого входной параметр это массив, то передать мы можем только массив, в то время как с ключевым словом params мы можем передать несколько значений, одно, ни одного или массив.

Использование ключевого слова params имеет свои ограничения: параметр params должен стоять последним при объявлении метода, если у метода есть другие параметры они должны находиться до параметра params.

**Область видимости**

Каждая переменная доступна в рамках определенного контекста или области видимость. Вне этого контекста переменная уже не существует.

Существуют различные контексты:

* Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса
* Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
* Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

Возьмём для примера нашу прошлую программу и немного дополним:

|  |
| --- |
| class Program  {  static int[] numbers = new int[] { 1, 2, 3, 4, 5, 25 }; // Контекст класса  static void Main(string[] args)  {  int[] intArray = new int[] { 7, 6, 5, 4, 3, 2, 1 };  int sumArray = TakeSumArray(intArray); // Контекст метода  int sumParams = TakeSumParams(1, 2, 3, 4, 5, 25); // Контекст метода  sumParams = TakeSumParams(1);  sumParams = TakeSumParams();  sumParams = TakeSumParams(numbers);  Console.ReadKey();  }  static int TakeSumParams(params int[] nums)  {  int sum = 0; // Контекст метода  Console.WriteLine(numbers.Length); // Массив numbers доступен внутри метода  foreach (int num in nums)  {  sum += num; // Переменная num имеет контекст блока кода  }  return sum;  }  static int TakeSumArray(int[] numbers) // этот массив переопределяет массив класса  {  int sum = 0; // Контекст метода  foreach (int num in numbers)  {  sum += num; // Переменная num имеет контекст блока кода  }  return sum;  }  static int TakeSum()  {  int sum = 0; // Контекст метода  for (int i = 0; i < numbers.Length; i++)  {  sum += numbers[i]; // Переменная i имеет контекст блока кода  }  return sum;  }  } |

Рассмотрим подробнее

Массив int[] intArray и переменные int sumArray, int sumParams доступны только в контексте метода Main.

Массив int[] numbers объявляется в теле класса и доступен во всех методах.

К примеру мы можем обратиться к нему в методе int TakeSumParams(params int[] nums).

Переменные num существуют только в теле цикла и имеют контекст блока кода, за пределами цикла обратиться к этим переменным невозможно, так же как и к переменной i цикла for.

В методе int TakeSumArray(int[] numbers) массив параметра скрывает одноимённый массив класса и внутри метода работает только с массивом контекста метода.

Фактически мы можем контекст ассоциировать с открывающими и закрывающими фигурными скобками.

**Рекурсивные функции**

Рекурсивная функция, это функция, которая построена так, что вызывает сама себя.

Рассмотрим классический пример рекурсивной функции: расчёт факториала

|  |
| --- |
| static int Factorial(uint x)  {  if (x == 0)  {  return 1;  }  else  {  return x \* Factorial(x - 1);  }  } |

Рассмотрим подробнее:

Тут задается условие, что если число не равно 0, то мы умножаем это число на результат этой же функции, в которую передаём число х – 1, т.е. происходит рекурсивный спуск. И так пока не дойдём до того момента пока х не станет равен 0.

В рекурсивной функции обязательно должно быть базовое ограничение (базовый вариант), в нашем случае это if (x == 0) return 1.

**Перечисления Enums**

Enums это Специальный тип в C#, ещё их называют перечисления. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum. Далее идет название перечисления, после которого указывается тип перечисления - он обязательно должен представлять целочисленный тип (byte, int, short, long). Если тип явным образом не указан, то по умолчанию используется тип int. Затем идет список элементов перечисления через запятую:

|  |
| --- |
| enum Days  {  Monday,  Tuesday,  Wednesday,  Thursday,  Friday,  Saturday,  Sunday  }  enum Time : byte  {  Morning,  Afternoon,  Evening,  Night  } |

Каждому элементу перечисления присваивается целочисленное значение, причём первый элемент будет иметь значение 0, второй 1 и т.д. Мы можем также явным образом указать значения элементов, либо указав значение первого элемента:

|  |
| --- |
| enum Days  {  Monday = 1,  Tuesday, // 2  Wednesday, // 3  Thursday, // 4  Friday, // 5  Saturday, // 6  Sunday // 7  } |

При этом каждый следующий элемент увеличивается на единицу от указанного.

Мы также можем явно указать номера для всех элементов перечисления, при этом значения могут иметь одинаковые значения, либо даже можно присваивать одной константе значение другой константы:

|  |
| --- |
| enum Color  {  White = 4,  Black = 2,  Green = 2,  Blue = White // Blue = 4  } |

Каждое перечисление фактически определяет новый тип данных. Затем в программе мы можем определить переменную этого типа и использовать ее:

|  |
| --- |
| enum Days  {  Monday,  Tuesday, // 2  Wednesday = 1, // 3  Thursday, // 4  Friday, // 5  Saturday, // 6  Sunday // 7  }  static void Main(string[] args)  {  Days day;  day = Days.Monday;  Console.WriteLine(day); // Monday  Console.ReadKey();  } |

В программе мы можем присвоить значение этой переменной. При этом в качестве ее значения должна выступать одна из констант, определенных в данном перечислении. То есть несмотря на то, что каждая константа сопоставляется с определенным числом, мы не можем присвоить ей числовое значение, например, Days day = 1;. И также если мы будем выводить на консоль значение этой переменной, то мы получим им константы, а не числовое значение. Если же необходимо получить числовое значение, то следует выполнить приведение к числовому типу:

|  |
| --- |
| Days day;  day = Days.Monday;  Console.WriteLine((int)day); // 0 |

Перечисления (в отличии от переменных) необязательно определять внутри класса, можно и вне класса, но в пределах пространства имён.

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия. Так, рассмотрим применение перечисления на более реальном примере:

|  |
| --- |
| class Program  {  enum Operation  {  Add = 1,  Subtract,  Multiply,  Divide  }  static void MathOp(double x, double y, Operation op)  {  double result = 0.0;  switch (op)  {  case Operation.Add:  result = x + y;  break;  case Operation.Subtract:  result = x - y;  break;  case Operation.Multiply:  result = x \* y;  break;  case Operation.Divide:  result = x / y;  break;  }  Console.WriteLine("Результат операции равен {0}", result);  }  static void Main(string[] args)  {  // Тип операции задаем с помощью константы Operation.Add, которая равна 1  MathOp(10, 5, Operation.Add);  // Тип операции задаем с помощью константы Operation.Multiply, которая равна 3  MathOp(11, 5, Operation.Multiply);  Console.ReadLine();  }  } |