АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Массивы

- Метод **Find<T> (T[] array, Predicate<T> match)** находит первый элемент, который удовлетворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
- **T? FindLast<T> (T[] array, Predicate<T> match)** находит последний элемент, который удовлетворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
- **int FindIndex<T> (T[] array, Predicate<T> match)** возвращает индекс первого вхождения элемента, который удовлетворяет определенному условию делегата match
- **int FindLastIndex<T> (T[] array, Predicate<T> match)** возвращает индекс последнего вхождения элемента, который удовлетворяет определенному условию
- **T[] FindAll<T> (T[] array, Predicate<T> match)** возвращает все элементы в виде массива, которые удовлетворяет определенному условию из делегата match
- **int IndexOf (Array array, object? value)** возвращает индекс первого вхождения элемента в массив
- **int LastIndexOf (Array array, object? value)** возвращает индекс последнего вхождения элемента в массив
- **void Resize<T> (ref T[]? array, int newSize)** изменяет размер одномерного массива

Массивы

- **void Reverse** располагает элементы массива в обратном порядке

Reverse(array, Int32, Int32)	Изменяет порядок подмножества элементов в одномерном массиве Array на обратный с определенного элемента в заданном количестве.
Reverse(Array)	Изменяет порядок элементов во всем одномерном массиве Array на обратный.
Reverse<T>(T[])	Изменяет порядок последовательности элементов в одномерном общем массиве на обратный.
Reverse<T>(T[], Int32, Int32)	Изменяет порядок последовательности подмножества элементов в одномерном общем массиве на обратный.

Массивы

- **void Sort** сортирует элементы одномерного массива

Sort(array, array, Int32, Int32, IComparer)	Сортирует диапазон элементов в паре одномерных объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя указанный интерфейс IComparer.
Sort(array, Int32, Int32, IComparer)	Сортирует элементы в диапазоне элементов одномерного массива Array, используя указанный интерфейс IComparer.
Sort(array, array, Int32, Int32)	Сортирует диапазон элементов в паре одномерных объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя реализацию интерфейса IComparable каждого ключа.
Sort(array, Int32, Int32)	Сортирует элементы в диапазоне элементов одномерного массива Array с помощью реализации интерфейса IComparable каждого элемента массива Array.
Sort(array, IComparer)	Сортирует элементы в одномерном массиве Array, используя указанный интерфейс IComparer.
Sort(array, array)	Сортирует пару одномерных объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя реализацию интерфейса IComparable каждого ключа.
Sort(array)	Сортирует элементы во всем одномерном массиве Array, используя реализацию интерфейса IComparable каждого элемента массива Array.
Sort(array, array, IComparer)	Сортирует пару одномерных объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя указанный интерфейс IComparer.

Массивы

Sort<T>(T[])	Сортирует элементы во всем массиве Array, используя реализацию универсального интерфейса IComparable<T> каждого элемента массива Array.
Sort<T>(T[], IComparer<T>)	Сортирует элементы в массиве Array, используя указанный универсальный интерфейс IComparer<T>.
Sort<T>(T[], Comparison<T>)	Сортирует элементы массива Array с использованием указанного объекта Comparison<T>.
Sort<T>(T[], Int32, Int32)	Сортирует элементы в диапазоне элементов массива Array, используя реализацию универсального интерфейса IComparable<T> каждого элемента массива Array.
Sort<T>(T[], Int32, Int32, IComparer<T>)	Сортирует элементы в диапазоне элементов массива Array, используя указанный универсальный интерфейс IComparer<T>.
Sort<TKey,TValue>(TKey[], TValue[])	Сортирует пару объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя реализацию универсального интерфейса IComparable<T> каждого ключа.
Sort<TKey,TValue>(TKey[], TValue[], IComparer<TKey>)	Сортирует пару объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя указанный универсальный интерфейс IComparer<T>.
Sort<TKey,TValue>(TKey[], TValue[], Int32, Int32)	Сортирует диапазон элементов в паре объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя реализацию универсального интерфейса IComparable<T> каждого ключа.
Sort<TKey,TValue>(TKey[], TValue[], Int32, Int32, IComparer<TKey>)	Сортирует диапазон элементов в паре объектов Array (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве Array, используя указанный универсальный интерфейс IComparer<T>.

Массивы

Поиск индекса элемента

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

// находится индекс элемента "Bob"
int bobIndex = Array.BinarySearch(people, "Bob");
// находится индекс первого элемента "Tom"
int tomFirstIndex = Array.IndexOf(people, "Tom");
// находится индекс последнего элемента "Tom"
int tomLastIndex = Array.LastIndexOf(people, "Tom");
// находится индекс первого элемента, у которого длина строки больше 3
int lengthFirstIndex = Array.FindIndex(people, person => person.Length > 3);
// находится индекс последнего элемента, у которого длина строки больше 3
int lengthLastIndex = Array.FindLastIndex(people, person => person.Length >
3);

Console.WriteLine($"bobIndex: {bobIndex}");                 // 2
Console.WriteLine($"tomFirstIndex: {tomFirstIndex}");       // 0
Console.WriteLine($"tomLastIndex: {tomLastIndex}");         // 4
Console.WriteLine($"lengthFirstIndex: {lengthFirstIndex}"); // 3
Console.WriteLine($"lengthLastIndex: {lengthLastIndex}");   // 5
```

Массивы

Поиск элемента по условию

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

//находится первый и последний элементы где длина строки больше 3 символов
string? first = Array.Find(people, person => person.Length > 3);
Console.WriteLine(first); // Kate
string? last = Array.FindLast(people, person => person.Length > 3);
Console.WriteLine(last); // Alice

// находятся элементы, у которых длина строки равна 3
string[] group = Array.FindAll(people, person => person.Length == 3);
foreach (var person in group) Console.WriteLine(person);
// Tom Sam Bob Tom
```

Массивы

Изменение порядка элементов массива

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

Array.Reverse(people);

foreach (var person in people)
    Console.Write($"{person} ");
// "Alice", "Tom", "Kate", "Bob", "Sam", "Tom"
```

Можно изменить порядок только части элементов:

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

// изменяется порядок 3 элементов начиная с индекса 1
Array.Reverse(people, 1, 3);

foreach (var person in people)
    Console.Write($"{person} ");
// "Tom", "Kate", "Bob", "Sam", "Tom", "Alice"
```

Массивы

Изменение размера массива

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

// уменьшение массива до 4 элементов
Array.Resize(ref people, 4);

foreach (var person in people)
    Console.Write($"{person} ");
// "Tom", "Sam", "Bob", "Kate"
```

Копирование массива

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

var employees = new string[3];

//копируются 3 элемента из массива people с индекса 1  и вставляются в массив employees начиная с индекса 0
Array.Copy(people,1, employees,0, 3);

foreach (var person in employees)
    Console.Write($"{person} ");
// Sam Bob Kate
```

Массивы

Сортировка массива

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

Array.Sort(people);

foreach (var person in people)
    Console.Write($"{person} ");

// Alice Bob Kate Sam Tom Tom
```

Перегрузка

```
var people = new string[] { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

// сортируются с 1 индекса 3 элемента
Array.Sort(people, 1, 3);

foreach (var person in people)
    Console.Write($"{person} ");

// Tom Bob Kate Sam Tom Alice
```

Методы

Метод - это именованный блок кода, который выполняет некоторые действия и содержит набор инструкций.
Общее определение методов выглядит следующим образом:

[модификаторы] тип_возвращаемого_значения название_метода ([параметры])

```
{
    // тело метода
}
```

Модификаторы метода:

public

Методы с модификатором public являются открытыми и как правило, устанавливают значения для поля или возвращают значения поля.

private

Модификатор private установлен по умолчанию, поэтому явно его объявлять не обязательно. Закрытые методы выполняют какие либо действия внутри класса.

protected

К методам с модификатором protected могут получить доступ только члены базового класса, где они определены и члены производного класса.

Методы

new

Модификатор new явно скрывает метод базового класса, который имеет такое имя, как и метод производного. Не желательно создавать имена членов в производном классе, которые соответствуют именам в базовом классе.

static

Модификатор static создает статический метод, который принадлежит только типу, а не его объекту.

virtual / override

Полиморфизм - это возможность определения для каждого производного класса собственного способа выполнения одного и того же метода, определенным в базовом классе.
Модификатор virtual используется в базовом классе и указывает, что метод может быть переопределен в производном классе.
Модификатор virtual нельзя использовать с модификаторами override, static, abstract, private.
Модификатор override используется в производном классе и указывает новую реализацию метода, унаследованного от базового класса.
Модификатор override нельзя использовать с модификаторами virtual, static, new.

Методы

abstract / override

Абстрактный метод создается с помощью модификатора abstract в абстрактном базовом классе. Он не имеет тела. Абстрактный метод может быть объявлен только в абстрактном классе. Реализация абстрактного метода должна происходить в производном классе.

Если код реализации метода написан не на языке C#, то такой метод называют внешним.

При объявлении внешнего метода используется модификатор extern, так же нужно указать в объявлении, что такой метод должен быть открытым public и статическим static.

Каждому объявлению метода предшествует атрибут DllImport.

Синтаксис:

```
[DllImport("имя_библиотеки.dll",CharSet = CharSet.Unicode)]
public static extern тип имя_метода(список_параметров);
```

Методы

```
using System;
class HelloWorld
{
static void SHello()
 {
Console.WriteLine("Hello");
 }
 static void Main()
 {
 SHello();
 SHello();
}
}
```

Методы

Определение метода

```
using System;
class HelloWorld
{
static void SHello()
 {
Console.WriteLine("Hello");
 }
 static void Main()
 {
 SHello();
 SHello();
}
}
```

Определен метод SHello, который выводит сообщение. К названиям методов предъявляются те же требования, что и к названиям переменных. Названия методов начинаются с большой буквы.

Перед названием метода идет возвращаемый тип данных.

После названия метода в скобках идет перечисление параметров.

После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций.

Методы

Сокращенная запись методов

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода.

```
void SayHello() => Console.WriteLine("Hello");
```

Параметры позволяют передать в метод некоторые входные данные. Параметры определяются через заятую в скобках после названия метода в виде:

тип_метода имя_метода (тип_параметра1 параметр1, тип_параметра2 параметр2, ...)

```
{
    // действия метода
}
```

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

Методы

```
using System;
class HelloWorld {

        static void Main()
        {
             PrintMessage("Hello work");
             PrintMessage("Hello ");
             PrintMessage("C#");
        }
           static void PrintMessage(string message)
            {
                Console.WriteLine(message);
            }
}
```

Формальные параметры и **фактические параметры**.

Формальные параметры - это собственно параметры метода (в данном случае message).

Фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Методы

Метод, который складывает два числа:

```
 static void Sum(int x, int y)
{
    int result = x + y;
    Console.WriteLine($"{x} + {y} = {result}");
}

Sum(10, 15);
```

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

Внутри метода вычисляется сумма переданных чисел и выводится на консоль.

При вызове метода Sum значения передаются параметрам по позиции. Ввызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y.

Методы

Также параметры могут использоваться в сокращеной версии метода:

```
static void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");

Sum(10, 15);
```

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

Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры.

Для таких параметров необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}
```

Методы

Параметры age и company являются необязательными, так как им присвоены значения. Поэтому при вызове метода можно не передавать для них данные:

```
static void PrintPerson(string name, int age = 1, string company = "Und")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

PrintPerson("Tom", 37, "Micr");
PrintPerson("Tom", 37);
PrintPerson("Tom");
```

Методы

Именованные параметры

При вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. То есть аргументы передавались параметрам **по позиции**.

Существует возможность нарушить подобный порядок, используя именованные параметры:

Для передачи значений параметрам по имени, при вызове метода указывается имя параметра и через двоеточие его значение: name:"Tom"

```
static void PrintPerson(string name, int age = 1, string company = "Und")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

PrintPerson("Tom", company:"Micr", age: 37);
PrintPerson(age:41, name: "Bob");
PrintPerson(company:"Google", name:"Sam");
```

Методы

Возвращение значения и оператор return

Метод может возвращать значение, какой-либо результат. Для этого применяется оператор return, после которого идет возвращаемое значение:

return возвращаемое значение;

```
using System;
class HelloWorld {

        static void Main()
        {
            Console.WriteLine(PrintMessage("Hello work"));
            Console.WriteLine(PrintMessage("Hello "));
            Console.WriteLine(PrintMessage("C#"));
        }
            static string PrintMessage(string message)
            {
                return message;
            }
}
```

Методы

Между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие.

Результат методов, который возвращают значение, можно присвоить переменным или использовать иным образом в программе:

```
using System;
class HelloWorld {
        static void Main()
        {
            string a,b,c;
            a=PrintMessage("Hello work");
            b=PrintMessage("Hello ");
            c=PrintMessage("C#");
            Console.WriteLine(a);
            Console.WriteLine(b);
            Console.WriteLine(c);
        }
            static string PrintMessage(string message)
            {
                return message;
            }
}
```

Методы

Можно передать в качестве значения параметру другого метода:

```
using System;
class HelloWorld {

        static void Main()
        {
            string a,b;
            a=GetMessage("Hello work");
            b=GetMessage("Hello ");

            Console.WriteLine(a);
            Console.WriteLine(b);
            PrintMessage(GetMessage("C#"));
        }
            static void PrintMessage(string message)
            {
                Console.WriteLine(message);
            }

            static string GetMessage(string message)
            {
                return message;
            }
}
```

Методы

Можно использовать оператор return и в методах с типом void.
В этом случае после оператора return не ставится никакого возвращаемого значения. Пример в зависимости от определенных условий произвести выход из метода:

```
static void PrintPerson(string name, int age)
{
    if(age > 120 || age < 1)
    {
        Console.WriteLine("Недопустимый возраст");
        return;
    }
    Console.WriteLine($"Имя: {name}  Возраст: {age}");
}

PrintPerson("Tom", 37);
PrintPerson("Dunkan", 1234);
```

Передача параметров по ссылке и значению. Выходные параметры

В языке C# существует два способа передачи параметров в метод; **по значению** и **по ссылке**.

Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров.

При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее работает с этой копией независимо от самой переменной.

Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор **ref**:

```
static void Increment(ref int n)
{
    n++;
    Console.WriteLine($"Число в методе Increment: {n}");
}

int number = 5;
Console.WriteLine($"Число до метода Increment: {number}");
Increment(ref number);
Console.WriteLine($"Число после метода Increment: {number}");
```

Передача параметров по ссылке и значению. Выходные параметры

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

Так, в метод Increment передается ссылка на саму переменную number в памяти. И если значение параметра n в Increment изменяется, то это приводит и к изменению переменной number, так как и параметр n и переменная number указывают на один и тот же адрес в памяти.

Модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

Выходные параметры. Модификатор out

Параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор **out**

```
static void Sum(int x, int y, out int result)
{
    result = x + y;
}
```

Передача параметров по ссылке и значению. Выходные параметры

Ключевое слово out используется как при определении метода, так и при его вызове.
Методы, использующие такие параметры, обязательно должны присваивать им определенное значение.

Входные параметры. Модификатор in

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором **in**. Модификатор **in** указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить.

Поля и методы встроенных типов

Любой встроенный тип C# построен на основе стандартного класса библиотеки .NET. Это значит, что у встроенных типов данных C# есть *методы и поля*. С помощью них можно, например, получить:

- **double.MaxValue** (или System.Double.MaxValue) - максимальное число типа double;
- **uint.MinValue** (или System.UInt32.MinValue) - минимальное число типа uint.

В вещественных классах есть элементы:

- положительная бесконечность **PositiveInfinity**;
- отрицательная бесконечность **NegativeInfinity**;
- «не является числом»: **NaN**.

Введение в исключения

- При вычислении выражений могут возникнуть ошибки (переполнение, деление на ноль).
- В C# есть механизм *обработки исключительных ситуаций* (*исключений*), который позволяет избегать аварийного завершения программы.
- Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью *выбрасывания (генерирования) исключения*.
- Каждому типу ошибки соответствует свое исключение. Исключения являются классами, которые имеют общего предка — класс Exception, определенный в пространстве имен System.
- Например, при делении на ноль будет выброшено исключение DivideByZeroException, при переполнении — исключение OverflowException.