Methods

.NET

In C#, every action is performed within a method. A method is a code block that contains a series of statements. A program calls the method, specifying any required method arguments.

https://learn.microsoft.com/en-us/dotnet/csharp/methods

Methods overview

https://learn.microsoft.com/en-us/dotnet/csharp/methods

The Main method is the entry point for every C# application. Methods are called by the ***common language runtime (CLR)*** when the program is started.

Methods are declared in a class, record, or struct and have unique signatures. They are invoked by other parts of the application that have access to the method.

```
1   namespace markshelloworld;
2
3   class Program
4   {
5       //the main method is the entry point of every C# program
6       static void Main(string[] args)
7       {
8           Console.WriteLine("Hello, Mark!");
9           int x = MyFunc(5);// you call a Method with arguments
10          Console.WriteLine(x);
11      }
12
```

Class Members Overview

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects

Members of a class are:

- Constructors - To initialize instances of the class
- Constants - Constant values
- Fields – Variables
- Methods – Computations/actions that can be performed
- Properties – Fields combined with the actions associated with reading/writing them
- Types - Nested types declared by the class

Class members can be:

- static - belong to classes. Envoked with: ClassName.MethodName();
- instance - belong to ***instances*** of classes. Envoked with: InstanceName.MethodName();

Methods

https://docs.microsoft.com/en-us/dotnet/csharp/methods

A method (procedure, function) is a code block that contains a series of statements. A program calls the method and includes any required arguments. Every C# command is executed within a method.

Methods are declared in a ***class*** or ***struct*** by specifying a method signature that contains:

- (optional) access level
- (optional) modifiers
- Return value
- Method name
- Method parameters

```
// 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; }
```

Methods

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#methods

There are two categories of methods:

- ***Static*** – accessed directly through the class
- ***Instance*** – accessed though instances of a class.

Methods have a ***Method Signature*** which consists of:

- The name of the method,
- The ***type*** parameters (if needed),
- Parameter names.

*The signature of a method doesn't include the return type.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```
// Methods
public void Add(T item)
{
    if (count == Capacity) Capacity = count * 2;
    items[count] = item;
    count++;
    OnChanged();
}
```

Static and Instance Methods

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#static-and-instance-methods

---

static method –

- declared with a ***static*** modifier.
- doesn't operate on a specific class instance.
- Only accessed through the class name. (Ex. MyClassName.MyStaticMethod())
- Cannot use this.

instance method –

- declared with any modifier other than ***static***.
- operates on a specific class ***instance*** only.
- can access both ***static*** and ***instance*** members.
- Can use this.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Method Invocation

https://docs.microsoft.com/en-us/dotnet/csharp/methods#method-invocation

There are two types of methods:

Instance and ***Static***

Instance methods

Require an object be instantiated to be called –

myClassInstance.doWork();

```
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);
   }
}
```

Method Invocation

https://docs.microsoft.com/en-us/dotnet/csharp/methods#method-invocation

There are two types of methods:

Instance and ***Static***

Static methods

Can be called without instantiating an object –
myClassName.doWork();

```
public class Example
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

Method Overloading

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#method-overloading

Method overloading

- permits multiple methods in the same class to have the same name
- Methods must each have unique ***parameter*** lists.
- The compiler uses ‘overload resolution’ to determine the specific method to invoke.
- ‘Overload resolution’ finds the one method that best matches the arguments or reports an error if none is found.
- A method can be selected by explicitly ***casting*** the arguments to the exact ***parameter*** types.

```
using System;
class OverloadingExample
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F<T>(T x)
    {
        Console.WriteLine("F<T>(T)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Value and reference Parameters

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

Parameters are used to receive variables from method calls.

There are five types of method parameters:

```
static void Divide(int x, int y,
{
    result = x / y;
    remainder = x % y;
}
```

1. value parameter

- a copy of the argument passed. Changes don't affect the original argument. Can be options by specifying a default value.

```
using System;
class RefExample
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void SwapExample()
    {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine($"{i} {j}");    // Outputs "2 1"
    }
}
```

2. reference parameter

- declared with the 'ref' modifier. Used for passing value arguments by reference. The argument must be a variable with a definite value. Changes take place on the original value.

out and params parameters

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params

---

- **3. output parameter** –
  - declared with the out modifier.
    - Used for passing arguments by reference.
    - An explicitly assigned value is not allowed before the method call.
- **4. parameter array** –
  - permits an 'N' number of arguments to be passed to a method.
  - Declared with the params modifier.
  - Must be the last parameter and be a 1-D array.
  - Write() and WriteLine() methods use parameter arrays.

```
using System;
class OutExample
{
    static void Divide(int x, int y, out int result, out int remainder)
    {
        result = x / y;
        remainder = x % y;
    }
    public static void OutUsage()
    {
        Divide(10, 3, out int res, out int rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }
}
```

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

in parameter

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

---

The in keyword causes arguments to be passed by reference but ensures the argument is not modified. It makes the formal parameter an alias for the argument, which must be a variable.

It is like the ref or out keywords, except that in arguments cannot be modified by the called method. Whereas ref arguments may be modified, out arguments must be modified by the called method, and those modifications are observable in the calling context.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

Optional Params and Default Values

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs

---

A parameter can be ***optional***. Any call must provide arguments for all required parameters but can omit arguments for ***optional*** parameters.

Each ***optional*** parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

Optional parameters are at the end of the parameter list after all required parameters. The caller must provide arguments for all required parameters before any optional parameters.