Cheatsheets / **Learn C#: Methods**

Learn C#: Methods

`Main()` method

In C#, the `Main()` method is the application's entry point. When the application is run, execution begins at the start of the `Main()` method.

Method Overloading

In C#, multiple methods can have the same name, as long as they each have a different method signature. A method's **signature** comprises its name, parameter types, and parameter order. Defining multiple methods with the same name is called **method overloading**.

```
// MultiplyValues has 2 overloads that accept different numbers of parameters
static int MultiplyValues(int a, int b)
{
  return a * b;
}

static int MultiplyValues(int a, int b, int c)
{
  return a * b * c;
}

static void Main(string[] args)
{
  MultiplyValues(2, 3); // Returns 6
  MultiplyValues(2, 3, 4); // Returns 24
}
```

Variables Inside Methods

Parameters and variables declared inside of a method cannot be used outside of the method's body. Attempting to do so will cause an error when compiling the program!

```
static void DeclareAndPrintVars(int x)
{
  int y = 3;
  // Using x and y inside the method is fine.
  Console.WriteLine(x + y);
}

static void Main()
{
  DeclareAndPrintVars(5);

  // x and y only exist inside the body of DeclareAndPrintVars, so we cannot use them here.
  Console.WriteLine(x * y);
}
```

Calling Methods

A method is called by using the method name followed by a set of parentheses `()`. Any argument values should be included between the parentheses, separated by commas.

```
// Console.WriteLine is a method, which we called using parentheses
// its string argument, "Hello World!" in this case, gets printed to the console
// and the argument is passed between the parentheses
Console.WriteLine("Hello World!");
```

Optional Parameters

In C#, methods can be given *optional parameters*. A parameter is optional if its declaration specifies a *default argument*. Methods with an *optional parameter* can be called with or without passing in an argument for that parameter. If a method is called without passing in an argument for the *optional parameter*, then the parameter is initialized with its default value.
To define an *optional parameter*, use an equals sign after the parameter declaration followed by its default value.

```
// y and z are optional parameters.
static int AddSomeNumbers(int x, int y =
3, int z = 2)
{
  return x + y + z;
}

// Any of the following are valid method
calls.
AddSomeNumbers(1); // Returns 6.
AddSomeNumbers(1, 1); // Returns 4.
AddSomeNumbers(3, 3, 3); // Returns 9.
```

What is a Method?

In C#, a **method** is a reusable set of instructions that performs a specific task. Methods can be passed one or more values as *inputs* and can *output* one or more values at the end of their execution.

Positional vs Named Arguments

In C#, an argument passed to a method may be identified by position or name.
Positional arguments are values passed directly to the method call in the same order as the defined parameters. **Named arguments** are values passed to the method call along with the name of the parameter they should be assigned to, irrespective of the order in which the parameters are defined. An argument can be named by using the parameter name followed by a colon (:), and then the argument value.
Positional and named arguments may be used within the same method call, but all positional arguments must precede all named arguments.

```
static void CalculateMean(double sumOfValues, double numberOfValues = 10)
{
  double result = sumOfValues / numberOfValues;
  Console.WriteLine(result);
}

static void Main(string[] args)
{
  // sumOfValues is passed positionally and numberOfValues is passed by name
  CalculateMean(50, numberOfValues: 20);
  // Prints 2.5

  // both arguments are passed by name
  CalculateMean(numberOfValues: 4, sumOfValues: 21);
  // Prints 5.25
}
```

Parameters and Arguments

In C#, a method can take inputs via *parameters*. Parameters are variables that can be used within the method body. When a method is called, the actual values passed to the method are called *arguments*.

```
// IntroduceSelf takes two parameters, name and age
static void IntroduceSelf(string name, int age)
{
  // name and age are used within the method body
  Console.WriteLine($"Hi, my name is {name}, and I am {age} years old.");
}

static void Main(string[] args)
{
  // "Astrid" and 28 are passed as arguments to the method call
  // and correlate to the name and age parameters respectively
  IntroduceSelf("Astrid", 28);
  // Prints "Hi, my name is Astrid, and I am 28 years old."
}
```

Void Return Type

In C#, methods that do not `return` a value have a `void` return type.

`void` is not an actual data type like `int` or `string`, as it represents the lack of an output or value.

```
// This method has no return value
static void DoesNotReturn()
{
  Console.WriteLine("Hi, I don't return like a bad library borrower.");
}
// This method returns an int
static int ReturnsAnInt()
{
  return 2 + 3;
}
```

Method Declaration

In C#, a *method declaration*, also known as a *method header*, includes everything about the method other than the method's body. The method declaration includes:

- the method name
- parameter types
- parameter order
- parameter names
- return type
- optional modifiers

```
// This is an example of a method header.
static int MyMethodName(int parameter1,
string parameter2) {
  // Method body goes here...
}
```

Return Keyword

In C#, the `return` statement can be used to return a value from a method back to the method's caller. When `return` is invoked, the current method terminates and control is returned to where the method was originally called. The value that is returned by the method must match the method's *return type*, which is specified in the *method declaration*.

```
static int ReturnAValue(int x)
{
  // We return the result of computing x
* 10 back to the caller.
  // Notice how we are returning an int,
which matches the method's return type.
  return x * 10;
}

static void Main()
{
  // We can use the returned value any
way we want, such as storing it in a
variable.
  int num = ReturnAValue(5);
  // Prints 50 to the console.
  Console.WriteLine(num);
}
```

Out Parameters

`return` can only return one value. When multiple values are needed, `out` parameters can be used. `out` parameters are prefixed with `out` in the method header. When called, the argument for each `out` parameter must be a *variable* prefixed with `out`. The `out` parameters become aliases for the variables that were passed in. So, we can assign values to the parameters, and they will persist on the variables we passed in after the method terminates.

```
// f1, f2, and f3 are out parameters, so they must be prefixed with `out`.
static void GetFavoriteFoods(out string f1, out string f2, out string f3)
{
  // Notice how we are assigning values to the parameters instead of using `return`.
  f1 = "Sushi";
  f2 = "Pizza";
  f3 = "Hamburgers";
}

static void Main()
{
  string food1;
  string food2;
  string food3;
  // Variables passed to out parameters must also be prefixed with `out`.
  GetFavoriteFoods(out food1, out food2, out food3);
  // After the method call, food1 = "Sushi", food2 = "Pizza", and food3 = "Hamburgers".
  Console.WriteLine($"My top 3 favorite foods are {food1}, {food2}, and {food3}");
}
```

Print Share