codecademy

Learn C#: Methods

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.
```

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);
}
```

Void Return Type

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

```
// This method has no return value
static void DoesNotReturn()
```

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

```
{
  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

A method declaration you've seen often is the declaration for the `Main` method (note there is more than one valid `Main` declaration):

```
static void Main(string[] args)
```

```
// 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}");
}
```

Expression-Bodied Methods

In C#, *expression-bodied methods* are short methods written using a special concise syntax. A method can only be written in *expression body* form when the method body consists of a single statement or expression. If the body is a single expression, then that expression is used as the method's return value. The general syntax is `returnType funcName(args...) => expression;`. Notice how "fat arrow" notation, `=>`, is used instead of curly braces. Also note that the `return` keyword is not needed, since the expression is implicitly returned.

```
static int Add(int x, int y)
{
  return x + y;
}

static void PrintUpper(string str)
{
  Console.WriteLine(str.ToUpper());
```

```
}

// The same methods written in
expression-body form.
static int Add(int x, int y) => x + y;

static void PrintUpper(string str) =>
Console.WriteLine(str.ToUpper());
```

Lambda Expressions

A *lambda expression* is a block of code that is treated like any other value or expression. It can be passed into methods, stored in variables, and created inside methods.
In particular, *lambda expressions* are useful for creating *anonymous methods*, methods with no name, to be passed into methods that require method arguments. Their concise syntax is more elegant than declaring a regular method when they are being used as one off method arguments.

```
int[] numbers = { 3, 10, 4, 6, 8 };
static bool isTen(int n) {
  return n == 10;
}

// `Array.Exists` calls the method passed
in for every value in `numbers` and
returns true if any call returns true.
Array.Exists(numbers, isTen);

Array.Exists(numbers, (int n) => {
  return n == 10;
});

// Typical syntax
// (input-parameters) => { <statements> }
```

Shorter Lambda Expressions

There are multiple ways to shorten the concise lambda expression syntax.

- The parameter type can be removed if it can be inferred.
- The parentheses can be removed if there is only one parameter.

As a side note, the usual rules for expression-bodied methods also apply to lambdas.

```
int[] numbers = { 7, 7, 7, 4, 7 };

Array.Find(numbers, (int n) => { return n
!= 7; });

// The type specifier on `n` can be
inferred based on the array being passed
in and the method body.
Array.Find(numbers, (n) => { return n !=
7; });

// The parentheses can be removed since
there is only one parameter.
```

```
Array.Find(numbers, n => { return n != 7;
});

// Finally, we can apply the rules for
expression-bodied methods.
Array.Find(numbers, n => n != 7);
```