Methods

What is a Method?

A method is a group of statements that perform a particular task.
In addition to the C# built-in methods, you may also define your own.

Methods have many advantages, including:
- Reusable code.
- Easy to test.
- Modifications to a method do not affect the calling program.
- One method can accept many different inputs.

> Every valid C# program has at least one method, the **Main** method.

Declaring Methods

To use a method, you need to **declare** the method and then **call** it.
Each method declaration includes:
- the return type
- the method name
- an optional list of parameters.

```
<return type> name(type1 par1, type2 par2, ... , typeN parN)
{
   List of statements
}
```

For example, the following method has an int parameter and returns the number squared:

```
int Sqr(int x)
{
  int result = x*x;
  return result;
}
```

The **return** type of a method is declared before its name. In the example above, the return type is **int**, which indicates that the method returns an integer value. When a method returns a value, it must include a **return** statement. Methods that return a value are often used in assignment statements.
Occasionally, a method performs the desired operations without returning a value. Such methods have a return type **void**. In this case, the method cannot be called as part of an assignment statement.

> **void** is a basic data type that defines a valueless state.

Calling Methods

Parameters are optional; that is, you can have a method with no parameters.
As an example, let's define a method that does not return a value, and just prints a line of text to the screen.

```
static void SayHi()
{
  Console.WriteLine("Hello");
}
```

Our method, entitled **SayHi**, returns **void**, and has no parameters.
To execute a method, you simply call the method by using the name and any required arguments in a statement.

```
static void SayHi()
{
  Console.WriteLine("Hello");
}

static void Main(string[] args)
{
  SayHi();
}
//Outputs "Hello"
```

Try It Yourself

The **static** keyword will be discussed later; it is used to make methods accessible in Main.

Calling Methods

You can call the same method multiple times:

```
static void SayHi()
{
  Console.WriteLine("Hello");
}

static void Main(string[] args)
{
  SayHi();
  SayHi();
  SayHi();
}
/* Outputs:
Hello
Hello
Hello
*/
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Parameters

Method declarations can define a list of **parameters** to work with.
Parameters are variables that accept the values passed into the method when called.
For example:

```
void Print(int x)
{
  Console.WriteLine(x);
}
```

This defines a method that takes one integer parameter and displays its value.

> Parameters behave within the method similarly to other local variables. They are created upon entering the method and are destroyed upon exiting the method.

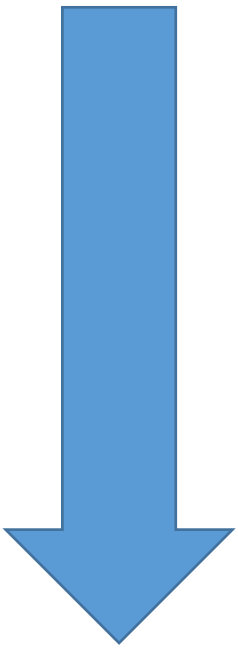
Parameters

Now you can call the method in Main and pass in the value for its parameters (also called **arguments**):

```
static void Print(int x)
{
  Console.WriteLine(x);
}
static void Main(string[] args)
{
  Print(42);
}
```

Try It Yourself

> The value 42 is passed to the method as an argument and is assigned to the formal parameter x.



Parameters

You can pass different arguments to the same method as long as they are of the expected type.
For example:

```
static void Func(int x)
{
  Console.WriteLine(x*2);
}
static void Main(string[] args)
{
  Func(5);
  //Outputs 10

  Func(12);
  //Outputs 24

  Func(42);
  //Outputs 84
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Multiple Parameters

You can have as many parameters as needed for a method by separating them with **commas** in the definition.
Let's create a simple method that returns the sum of two parameters:

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

The **Sum** method takes two integers and returns their sum. This is why the return type of the method is **int**. Data **type** and **name** should be defined for each parameter.

Methods return values using the **return** statement.

Multiple Parameters

A method call with multiple parameters must separate arguments with **commas**.
For example, a call to **Sum** requires two arguments:

```
static void Main(string[] args)
{
  Console.WriteLine(Sum(8, 6));
  // Outputs 14
}
```

Try It Yourself

In the call above, the return value was displayed to the console window. Alternatively, we can assign the return value to a variable, as in the code below:

```
static void Main(string[] args)
{
  int res = Sum(11, 42);
  Console.WriteLine(res);
  //Outputs 53
}
```

Try It Yourself

You can add as many parameters to a single method as you want. If you have multiple parameters, remember to separate them with **commas**, both when declaring them and when calling the method.

Optional Arguments

When defining a method, you can specify a **default value** for optional parameters. Note that optional parameters must be defined after required parameters. If corresponding arguments are missing when the method is called, the method uses the default values.
To do this, assign values to the parameters in the method definition, as shown in this example.

```
static int Pow(int x, int y=2)
{
  int result = 1;
  for (int i = 0; i < y; i++)
  {
    result *= x;
  }

  return result;
}
```

The Pow method assigns a default value of 2 to the y parameter. If we call the method without passing the value for the y parameter, the default value will be used.

```
static void Main(string[] args)
{
  Console.WriteLine(Pow(6));
  //Outputs 36

  Console.WriteLine(Pow(3, 4));
  //Outputs 81
}
```

Try It Yourself

As you can see, default parameter values can be used for calling the same method in different situations without requiring arguments for every parameter.
Just remember, that you must have the parameters with default values at the **end** of the parameter list when defining the method.

Named Arguments

Named arguments free you from the need to remember the order of the parameters in a method call. Each argument can be specified by the matching parameter name.
For example, the following method calculates the area of a rectangle by its height and width:

```
static int Area(int h, int w)
{
  return h * w;
}
```

When calling the method, you can use the parameter names to provide the arguments in any order you like:

```
static void Main(string[] args)
{
  int res = Area(w: 5, h: 8);
  Console.WriteLine(res);
  //Outputs 40
}
```

Try It Yourself

Named arguments use the **name** of the parameter followed by a **colon** and the **value**.

Passing Arguments

There are three ways to pass arguments to a method when the method is called: By **value**, By **reference**, and as **Output**.

By **value** copies the argument's value into the method's formal parameter. Here, we can make changes to the parameter within the method without having any effect on the argument.

By default, C# uses call **by value** to pass arguments.

The following example demonstrates by value:

```
static void Sqr(int x)
{
  x = x * x;
}
static void Main()
{
  int a = 3;
  Sqr(a);

  Console.WriteLine(a); // Outputs 3
}
```

Try It Yourself

In this case, **x** is the parameter of the **Sqr** method and **a** is the actual argument passed into the method.

As you can see, the **Sqr** method does not change the original value of the variable, as it is passed by **value**, meaning that it operates on the **value**, not the actual variable.

Passing by Reference

Pass by **reference** copies an argument's memory address into the formal parameter. Inside the method, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
To pass the value by reference, the **ref** keyword is used in both the call and the method definition:

```
static void Sqr(ref int x)
{
  x = x * x;
}
static void Main()
{
  int a = 3;
  Sqr(ref a);

  Console.WriteLine(a); // Outputs 9
}
```

Try It Yourself

The **ref** keyword passes the memory address to the method parameter, which allows the method to operate on the actual variable.

> The **ref** keyword is used both when defining the method and when calling it.

Passing by Output

Output parameters are similar to reference parameters, except that they transfer data out of the method rather than accept data in. They are defined using the **out** keyword.
The variable supplied for the output parameter need not be initialized since that value will not be used. Output parameters are particularly useful when you need to return multiple values from a method.
For example:

```
static void GetValues(out int x, out int y)
{
  x = 5;
  y = 42;
}
static void Main(string[] args)
{
  int a, b;
  GetValues(out a, out b);
  //Now a equals 5, b equals 42
}
```

Try It Yourself

Unlike the previous reference type example, where the value 3 was referred to the method, which changed its value to 9, output parameters get their value from the method (5 and 42 in the above example).

> Similar to the **ref** keyword, the **out** keyword is used both when defining the method and when calling it.

Overloading

Method **overloading** is when multiple methods have the **same name**, but **different parameters**.
For example, you might have a **Print** method that outputs its parameter to the console window:

```
void Print(int a)
{
  Console.WriteLine("Value: "+a);
}
```

The + operator is used to concatenate values. In this case, the value of **a** is joined to the text "Value: ".
This method accepts an integer **argument** only.
Overloading it will make it available for other types, such as **double**:

```
void Print(double a)
{
  Console.WriteLine("Value: "+a);
}
```

Now, the same **Print** method name will work for both integers and doubles.

Overloading

When overloading methods, the definitions of the methods must differ from each other by the types and/or number of parameters.
When there are overloaded methods, the method called is based on the arguments. An **integer** argument will call the method implementation that accepts an **integer** parameter. A **double** argument will call the implementation that accepts a **double** parameter. Multiple arguments will call the implementation that accepts the same number of arguments.

```
static void Print(int a) {
  Console.WriteLine("Value: " + a);
}
static void Print(double a) {
  Console.WriteLine("Value: " + a);
}
static void Print(string label, double a) {
  Console.WriteLine(label + a);
}

static void Main(string[] args) {
  Print(11);
  Print(4.13);
  Print("Average: ", 7.57);
}
```

Try It Yourself

You cannot overload method declarations that differ only by return type.
The following declaration results in an **error**.

```
int PrintName(int a) { }
float PrintName(int b) { }
double PrintName(int c) { }
```

Recursion

A **recursive** method is a method that calls itself.
One of the classic tasks that can be solved easily by recursion is calculating the **factorial** of a number.
In mathematics, the term **factorial** refers to the product of all positive integers that are less than or equal to a specific non-negative integer (n). The factorial of n is denoted as **n!**
For example:

> 4! = 4 * 3 * 2 * 1 = 24

> A recursive method is a method that calls itself.

Recursion

As you can see, a factorial can be thought of as repeatedly calculating num * num-1 until you reach 1.
Based on this solution, let's define our method:

```
static int Fact(int num) {
  if (num == 1) {
    return 1;
  }
  return num * Fact(num - 1);
}
```

In the **Fact** recursive method, the **if** statement defines the exit condition, a base case that requires no recursion. In this case, when **num** equals one, the solution is simply to return 1 (the factorial of one is one).
The recursive call is placed after the exit condition and returns **num** multiplied by the factorial of n-1.
For example, if you call the **Fact** method with the argument 4, it will execute as follows:
return 4*Fact(3), which is 4*3*Fact(2), which is 4*3*2*Fact(1), which is 4*3*2*1.

Now we can call our Fact method from Main:

```
static void Main(string[] args)
{
  Console.WriteLine(Fact(6));
  //Outputs 720
}
```

Try It Yourself

> The factorial method calls itself, and then continues to do so, until the argument equals 1. The exit condition prevents the method from calling itself indefinitely.

Making a Pyramid

Now, let's create a method that will display a pyramid of any height to the console window using star (*) symbols.
Based on this description, a parameter will be defined to reflect the number of rows for the pyramid.
So, let's start by declaring the method:

```
static void DrawPyramid(int n)
{
  //some code will go here
}
```

DrawPyramid does not need to return a value and takes an integer parameter **n**.
In programming, the step by step logic required for the solution to a problem is called an **algorithm**. The algorithm for MakePyramid is:
1. The first row should contain one star at the top center of the pyramid. The center is calculated based on the number of rows in the pyramid.
2. Each row after the first should contain an odd number of stars (1, 3, 5, etc.), until the number of rows is reached.

Based on the algorithm, the code will use for loops to display spaces and stars for each row:

```
static void DrawPyramid(int n)
{
 for (int i = 1; i <= n; i++)
 {
  for (int j = i; j <= n; j++)
  {
   Console.Write(" ");
  }
  for (int k = 1; k <= 2 * i - 1; k++)
  {
   Console.Write("*" + " ");
  }
  Console.WriteLine();
 }
}
```

Try It Yourself

The first **for** loop that iterates through each row of the pyramid contains two **for** loops.
The first inner loop displays the spaces needed before the first star symbol. The second inner loop displays the required number of stars for each row, which is calculated based on the formula (2*i-1) where i is the current row.
The final **Console.WriteLine();** statement moves the cursor to the next row.

> Now, if we call the **DrawPyramid** method, it will display a pyramid having the number of rows we pass to the method.

Output:

End.